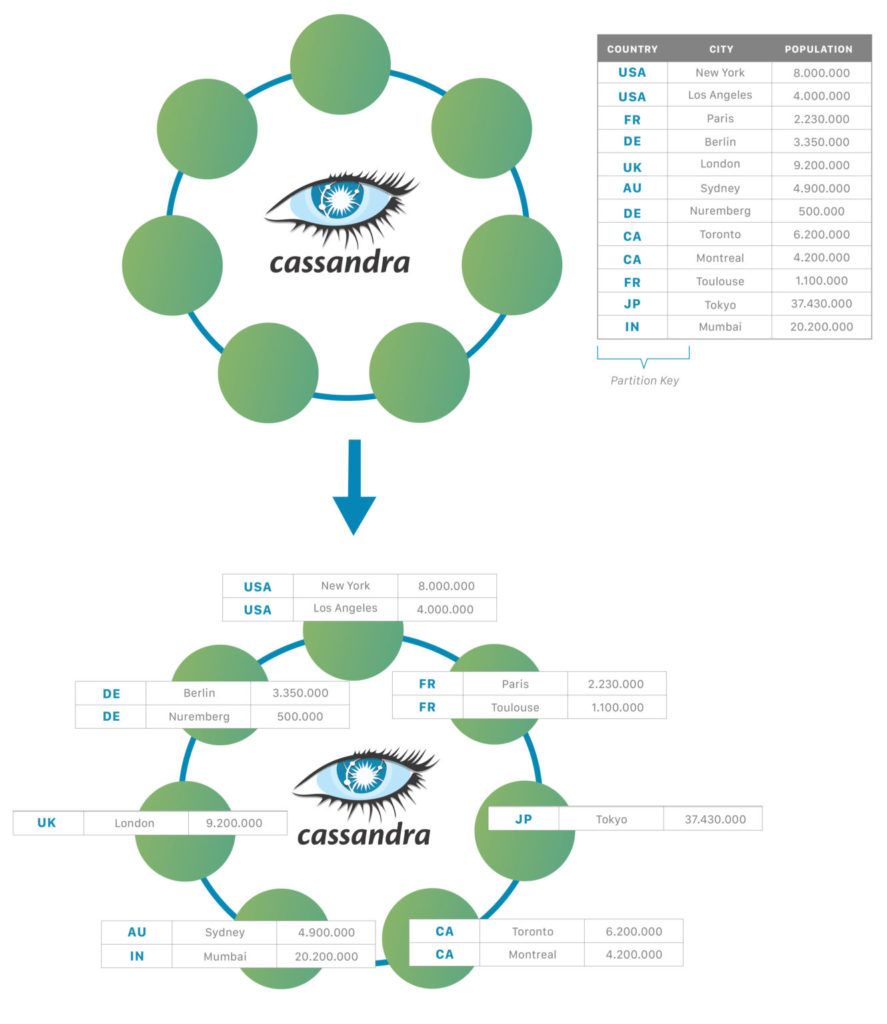
**Cassandra Overview**

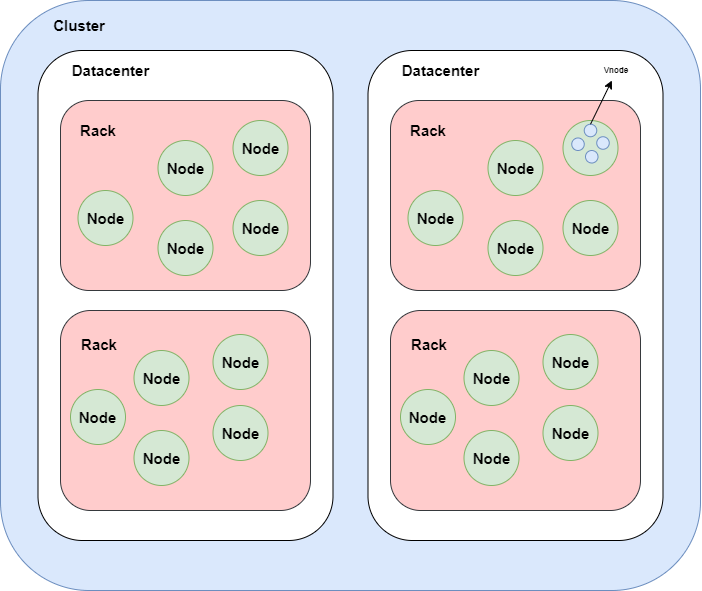
[Apache Cassandra](https://cassandra.apache.org/) is a NoSQL, distributed database management system. **The main advantage of Cassandra is that it can handle a high volume of structured data across commodity servers. Moreover, it provides high availability and provides no single point of failure.** Cassandra achieves this by using a **ring-type architecture**, where the smallest logical unit is a node. It uses the partitioning of data for optimizing queries.

Every piece of data has a partition key. The partition key for every row is hashed. As a result, we'll get a unique token for every piece of data. For each node, there is an assigned range of tokens. Consequently, data with the same token is stored on the same node. The [ring architecture](https://cassandra.apache.org/cassandra-basics) of nodes is shown below:

[](https://www.baeldung.com/wp-content/uploads/2021/06/apache-cassandra-diagrams-02-1330x1536-1.jpg)

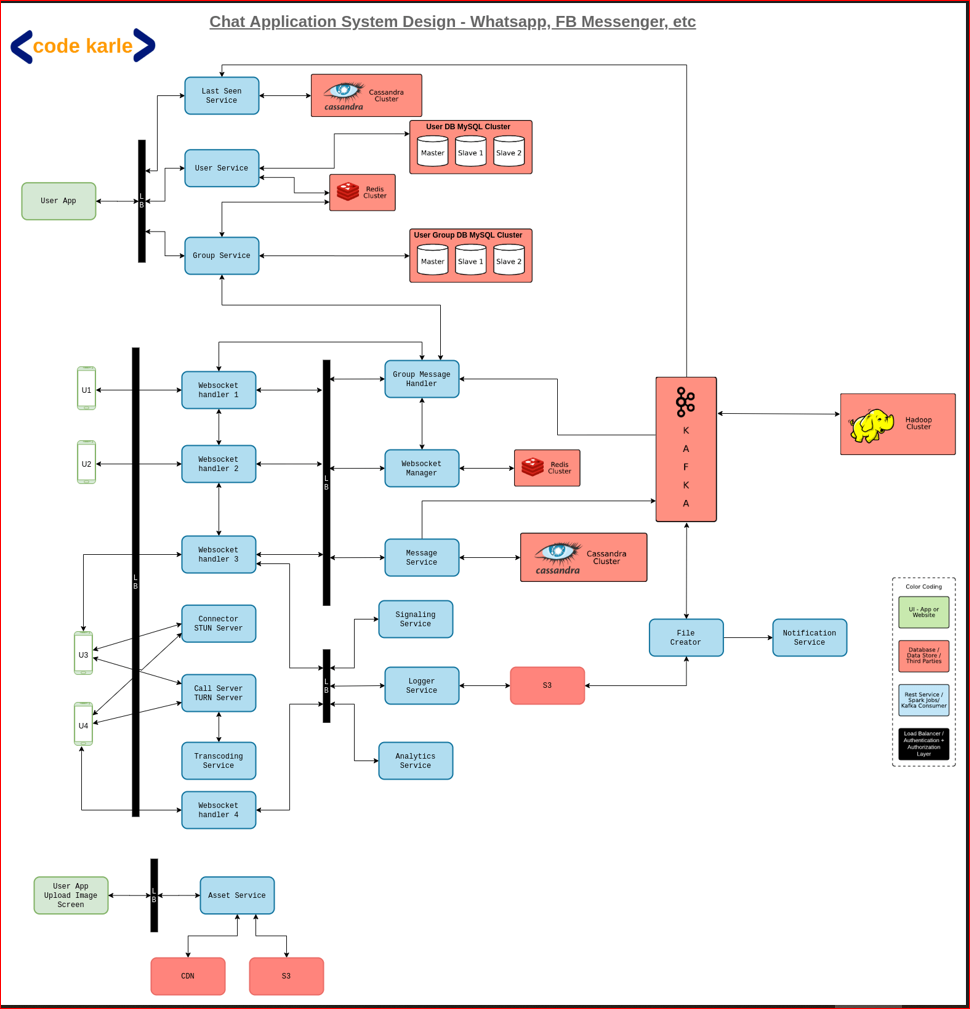
Cassandra Cluster

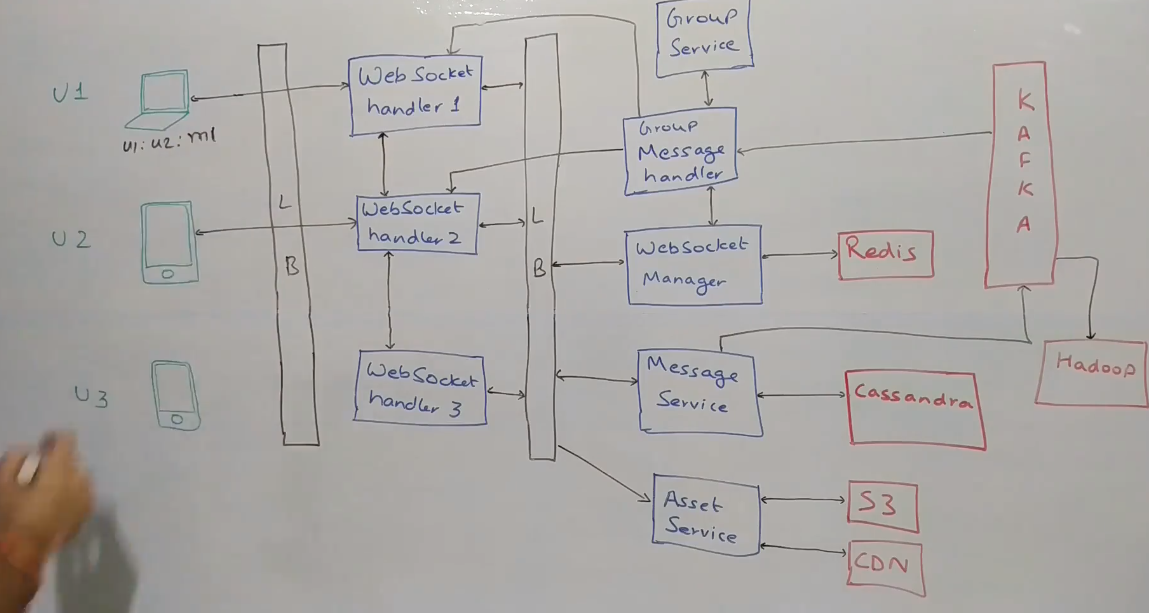
A cluster is a component that contains one or more datacenters. It's the most outer storage container in the database. One database contains one or more clusters. The hierarchy of elements in the Cassandra cluster is:

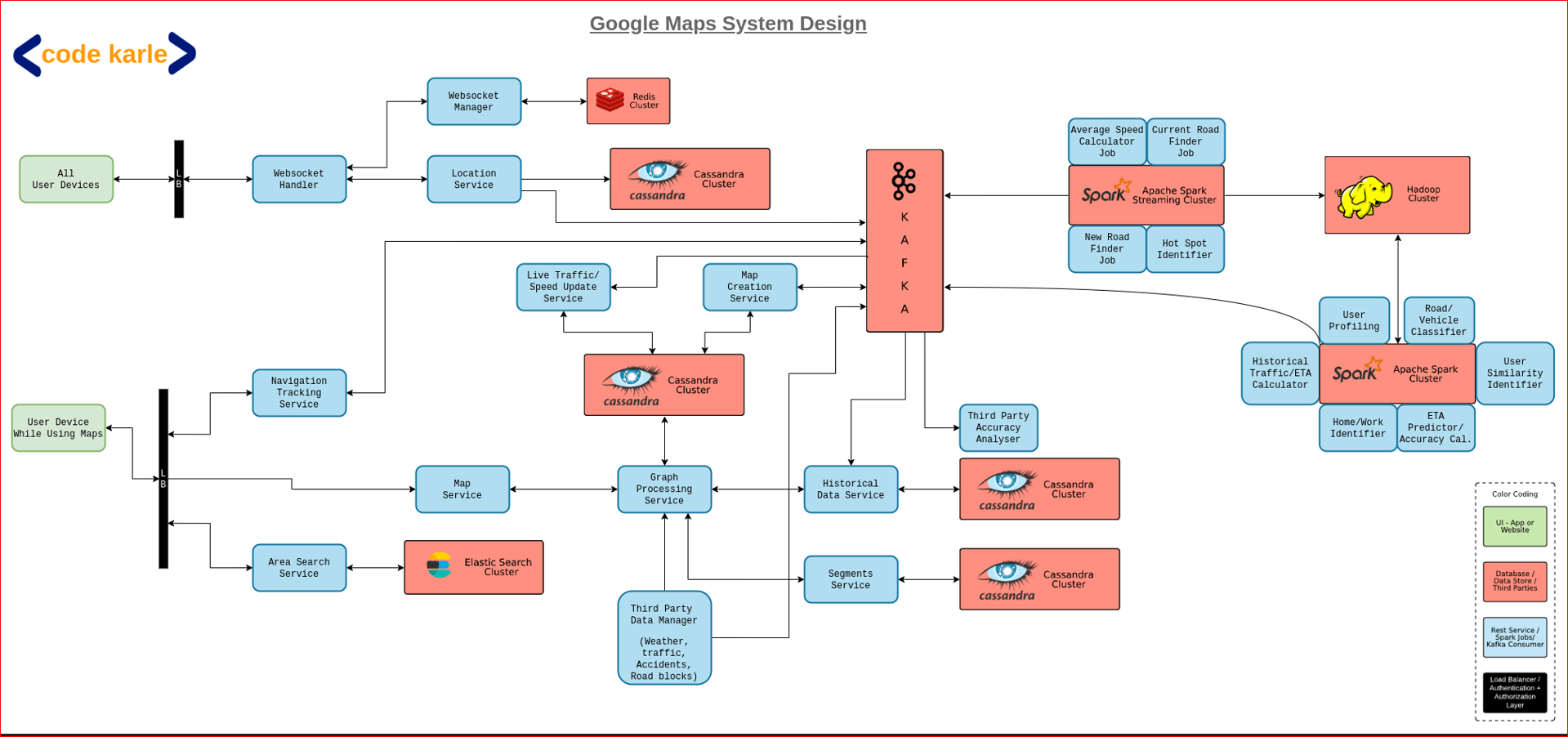
[](https://www.baeldung.com/wp-content/uploads/2021/06/cluster2.png)

First, we have clusters that consist of datacenters. Inside of datacenters, we have nodes that contain by default 256 virtual nodes.

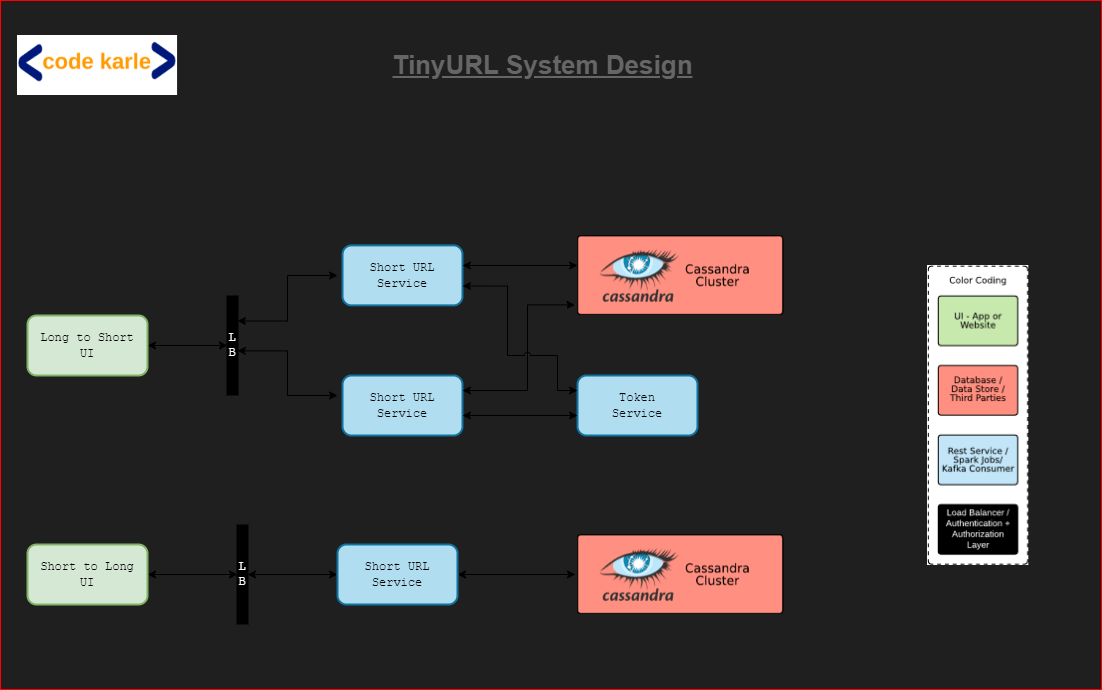
1. Chat app - Whatsapp, FB, Messenger System Design

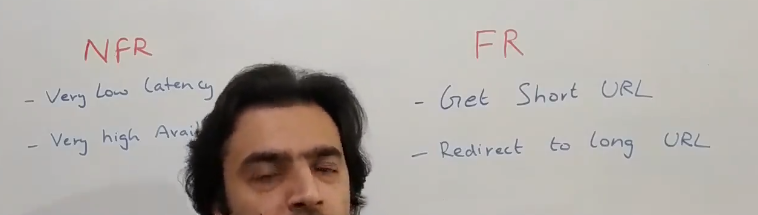




2. Google Maps System Design

3. TinyURL - URL Shortner System Design





Hello everyone! Welcome to codeKarle. In this video we'll be talking about possibly one of the most common system design interview question. So let's see how do we design a URL shortening service, something very similar to tinyurl.com. Let's start with the functional (FRs) and non functional requirements (NFRs). This is a fairly straightforward problem. You have these two functional requirements - given a long URL, you get a short URL. So, when somebody wants to shorten the URL, they'll give you a long URL and you need to return a shorter URL for that. On the other side, when somebody hits a short URL then you basically redirect the person to the longer URL, which is the second point. On the non-functional side, this service needs to be highly available and it should work at a very low latency. Because, let's just say, we were building it for a Facebook or a Twitter kind of a scenario, it would just be a bad experience if it just takes a lot of time to respond. And this being a very platform kind of a component, it needs to be always available and respond back in a very short time. The very first thing to answer is - What should be the length of our short URL? That depends upon the scale that we need to build this system for. If we're just told a couple of hundreds of URLs, maybe 2-3 characters are more than enough. But if you are building it at a Google scale or a Facebook scale, then we need to think over what should be the length of our short URL.

Because we should not run out of the short URLs. Another option is - we build a system agnostic of the length and we start from somewhere and it keeps on incrementing.

But let's start with a fixed length to start.

The very first question you need to ask your interviewer is - What is the traffic you are looking at? How many unique URLs will come to you for shortening? Probably every day, every month, something of some number. Until what duration you need to support that? Let's just assume that -

Once a URL is converted into a shorter URL, you need to save it for at least next ten years. Sounds a bit too much, but let's just take it as a number, okay. With that in mind, let's look at how can we come up with the potential length. So, let's just say that your interviewer tells you that there are 'X' number of requests that come to you every second. So, (X \* 60) is the number of requests. These are basically the number of short URLs that you have to generate.

So, these are the number of requests that come to you in a minute. These are the number of requests that come to you in an hour. Multiplying by 24 gives you the number of requests in a day. Multiplying by 365 gives you the number of requests in a year. Multiplying by 10 years. Whatever this number would be, you should be able to build a system which can handle these number of unique requests. Let's just say this number is 'Y'. Now, the next question to ask is -

What all characters can you include in the short URL? Should it be just numbers? Could it be alphabets? Could it be capital alphabets and small case alphabets? What all is allowed?

Generally, people take [A to Z], [a to z] and numbers. So maybe we can stick to that but you could ask your interviewer and then you could come up with a set of, basically a character set that is allowed for short URLs. For now, let us assume that we'll take [a to z], [A to Z] and numbers [0 to 9]. These are 62 characters which you can use in the short URL. Now, you need to have some length, you need to come up with a length which can handle these number of URLs, these number of unique combinations of these 62 characters. Let's just say if the length was 1, then you can support 62 URLs. Because there is one possibility each.

If the length of the short URL is 2, then you can support (62 ^ 2), which is some number let's say. Basically, you need to come up with a number where (62 ^ n) is greater than this number Y.

You solve this equation and you get the answer to n. n would be log62(Y). The feeling of it. Let's say, if this number comes down to be 4.5, then you take n = 5 so you will create short URLs of length 5. As a general number, (62 ^ 6) is somewhere around 58 billion if I am not wrong and (62 ^ 7) is somewhere around 3.5 trillion. Both of these are very massive numbers. Depending upon what is your use case, you can come up with some length. Let's just say, for our use case, we will stick to 7 characters. Generally, I have seen it's a standard that these short URLs generally use 7 characters, may be more at times. But we will stick to 7 characters and we will be able to support 3.5 trillion unique URLs. That I am assuming is more than this number (Y). If it so happens that this Y is even more, then you can probably take 8 characters, 9 characters or whatever. Now, let's look at one of the potential architectures that can solve this problem.

This is not a final version. We will iterate over it. So, let's just say, there is this UI, which takes the long URL as input and gives back the short URL. So, it will call the Short URL Service. There'll be multiple instances of this service, which somehow generates a short URL and stores it in the database and returns the short URL to the customer.

On the other side, this could also be used when somebody hits a short URL wherein the Short URL Service would fetch the long URLs from the database and then redirect the user to the longer URL.

This looks nice. But then we have not covered one important point that how does this service really generate a short URL. Even though we had those 62 characters, for all the communication further we'll just assume, we'll just use numbers so that it's easy to calculate and think over it. Let's just say, there are multiple instances of this service and all of them are getting some requests. This service could also generate a number 111, this service could also generate a number 111 and then we will end up with a collision. If two services generate a same short URL for two requests then we have a problem. In technical terms it's called a collision but for us it's a problem because we cannot have one short URL pointing to two long URLs. You can say that the possible solution could be that we first check in the database and then you know kind of retry. That would be fine but then the problem is that's not really efficient. What we need is a predictable way to generate a short URL knowing that there would be no collisions at all. One very simple way to do that is using one of the features of Redis. So let's just say, we use a Redis cluster and all of these services request a number from Redis. Redis makes sure that it will always return a unique number. So basically what it will do is - it will start the counting from 1 all the way up to billions, trillions, whatever. And each time it gets a request, it increments the counter and responds back. We will make sure that all of these guys are getting unique numbers and from unique number at base 10, they can convert to base 62 and generate the short URL. Now that would work fine but then we have a problem. The problem is that first of all, everybody now is connecting to Redis. So, we have Redis under a huge amount of load.

Second problem is - remember one of the important key things of any system design interview, you should never create a single point of failure. What is happening is - in all the requests that have to generate a short URL, this Redis becomes a single point of failure and there is no backup to it. If this goes down then there is no way we can recover, which is a problem.

You can argue that we can have multiple Redis but that's also tricky. Moreover, if the scale becomes beyond the capacity of one Redis machine, then we are in much bigger problem. Why? Because, let's say, if one Redis is not able to scale to the latency requirements that we wanted to, this will again start choking the whole system. Another alternate is - that we keep multiple Redis that will give us high availability and it will give us better performance because now there are multiple machines and some of these machines will talk to this Redis and let's say this machine talks to this Redis. This would work just fine till the time these Redis don't start generating duplicate numbers.

If both the Redis are starting from the same index, again they'll start generating duplicates. What we need to do is - somehow make sure that both these Redis do not generate a same number.

One very simple way is - to give a series to this Redis, another series to this Redis to start with. That would work fine. But, what if you have to add a third Redis. Now it becomes complicated. Now you need somebody to manage what series are being given to whom. Plus once you have that management piece, we might as well try to get away from Redis. Let's try to look at a slightly more advanced system which can generate unique URLs without using a Redis, without having a single point of failure. Let's look at one of the possible solutions which can solve this problem. Remember, what we need. Whenever we get a long URL and we need to convert it to a short URL, we basically need to make sure that our service machine should be able to generate a unique number, which will never ever get repeated even by any other instance of the service. So, each of these two service instances should generate a number, which other ones will never ever generate. Convert that number from base 10 to base 62 and return back. This is the flow that we need.

What we've done is - again going over the UI, so this is the, green thing is basically the Long to Short UI, from where somebody can give a long URL and expect to get a short URL as an output. It talks to a load balancer and behind it are multiple instances of Short URL Service. What I am trying to now do is - I have introduced a Token Service. The idea is all of these guys need to have a number. And making sure that none of them, none of the other machines generate the same number.

One of the simple ways to do that is -assign ranges to each of the machines. I can say that - this Short URL Service will request a token range from Token Service. Each time Token Service gets a request, it will run on a single threaded model

and it will give a range to any of the service instances. Let's say, it says that, I am giving a range (1 - 1,000) to this machine. Now, this machine on startup will also call Token Service. So, all of these services call Token Service either when they start up or when they are running out of the range.

Now, let's say, this service got a range from (1001 - 2000). Similarly, other instances of the same service would also get some range. This Token Service can be built on top of a MySQL, because it runs at a very low scale. It will get a call once in a couple of hours and it will return like the ranges. I have taken these numbers as (1 - 1,000). Realistically, it will be a bit larger number so that at least it can run for an hour or so. So, now Token Service has given numbers to each of these services. Now, the flow is very simple. Let's say, a request reaches this service. It takes the first number 1001, converts that into base 62 and

returns the output. Obviously after saving it. The next request comes, it increments it. It takes 1002, saves it in Cassandra, returns back. Next request comes in, it uses 1003. Let's say, it somehow gets lots of requests, gets to 2000. At that point in time, maybe a bit before that also, when we are closing the end of the range, it will query Token Service and get the next range. Let's say, the next range that this service got is (5001 - 6000), let's say. So, now it will continue processing this one. Now, when (5001 - 6000) is assigned to this service, when this service will make a request, it can possibly get (6001 - 7000), but it will never get this (5001 - 6000) range. Why? Because that thing is maintained by Token Service and the way it would probably work is - it will keep a range as a record and keep a flag, whether it is assigned or not. In a simple MySQL, on transaction basis, you get a record. When you get a request, you take the first unassigned token range and return that. And because it will then sit on top of a MySQL database, it will make sure that it is always unique. So, this becomes kind of our read flow.

Now, let's say, there is a massive amount of traffic. All we need to do is - keep multiple instances of Token Service, at max. Even though it might... so, there are multiple ways to scale it. First of all, we can increase the length of the range so that it doesn't get bombarded too often. So, instead of thousands - thousands, we can allocate millions of tokens at once so it doesn't get too many requests. And anyway, this service will be distributed in multiple geographies at least and multiple data centers so that this doesn't become a single point of failure.

There are some problems in this. Possible problems are - What if this service got this (5001 - 6000) range, started working for a while, used a couple of tokens and then it kind of got shut down and died? Let's say, there was an Out Of Memory error and this process got killed. What happens to those tokens? So, those tokens go away. Go away as in, there is no track record of how many tokens are being used. All what is happening is - it is iterating over that list in-memory and assigning one token to each request. If this service dies down, it will probably, on the same machine, it will get started up again and the next time we will get another token range. Let's say, third time, it gets a token range of (9001 - 10000). Now, all the unused tokens of this (5001 - 6000) range, there is no record of them. So, they are kind of gone forever. If you look at it, let's say, even if this happens a couple of times in a day, overall how many tokens? If with the length of 7 for a short URL, we are able to support 3.5 trillion unique numbers. That is a very massive number. If you lose out a couple of thousands of tokens out of this range, it is like taking a bucket out of an ocean.

It doesn't really matter. So, even if you lose out some tokens, it's okay!! But, if you start tracking those tokens, then it will become a fairly complicated system. Just to make it easy for us to develop, we will say that - okay! when the system shuts down, all those tokens go away. We don't need them. We will get a new range of tokens and work out of that. That is one thing that we could do. This is basically the long URL to short URL path. On the other side, when a short URL is hit, all you do is - you get a request in short URL, you hit your database and you get the longer URL, you send it back to this service. This service does a redirect and the user gets redirected to the main URL. One thing I have not answered is - Why did I use a Cassandra? Ideally, I could have used any database here. All we need to do is - to keep a database which can handle these many number of unique URLs. A MySQL, at these number of records would start giving some problems.

We can shard it probably and make it work. Cassandra will, for sure, work. So, that's the reason I've kept Cassandra. You can try using a MySQL here. With enough sharding and all of that, that would possibly also work fine. That's not that big of a problem. Plus this Token service is also using a MySQL. So, that could be shared across both the services as well. Whatever we have built till now is good enough from a functional and non-functional standpoint and it does what we were supposed to build. But is it good enough? Definitely not!! Why? Because this system, what we have built, does not give us any metrics about how it is being used, what kind of URLs are the top most used URLs, what kind of geographies people come from and it does not give any kind of metrics that we would want to have out of this system.

For example, whenever a person is generating a short URL, wouldn't it be nice, if we can tell them what kind of geography do your people come from or what kind of hits are you getting or what kind of user agents or devices the people are connecting from. All of those things would be valuable information for any person who's generating a short URL. Similarly for us, let's say, if we have multiple data centers.

Let's say, for example, we have 4 data centers and we want to choose any 2 of them as primary and 2 of them as standby data centers - and they are spread across the globe. If you have to decide which ones should be primary and which ones should be secondary, normally, what companies do is - they just make a decision based on somebody's gut feeling. But we could make a data-driven decision over here, depending upon what kind of traffic we are getting, from what kind of geographies and wherever we are getting most amount of traffic from, the data centers closer to those geographies could be made as primary data centers. For doing all of these things, it will be good to have a good enough amount of analytics that this system emits out.

Let's look at how can we do that? Let's see how the analytics is being built. The normal flow, the way it works is - whenever we get a request with a short URL, let's say, a client sends us a request. We query our database, get the longer URL from the database based on the short URL and return it back to the client, which then redirects the user to the longer URL.

Instead of just simply returning the URL, what we'll essentially do is - each time the request comes to us, that request will come with a lot of attributes. It will give us some origin header saying what is the platform from where this request has come in. So, let's just say, we posted this link on a Facebook or a LinkedIn kind of a platform, it will have that information. It will also have information about the user agent which is calling, let's say, if it's an Android application or iOS application or a browser. It will have that kind of an information as well and it will also have a source IP address.

All of these information, before sending out the response, we will also put that into a Kafka, which will be used to then power the Analytics. Based on the IP address, we'll also be able to come up with what country it is. So, all of those things can be taken care of on the Analytics side. But if we start putting into Kafka on each request when we get the request, it will impact our non-functional requirement of latency. Why? Because, now we are doing an additional step in the process and that will increase the latency. So, I would not recommend doing that. What instead we could do as a fairly nice solution is - make that Kafka request as a parallel call. So, you can have a different thread, in which you can send the Kafka write and return back to the user. And asynchronously, it can be returned to Kafka.

What is the problem in doing an asynchronous thing? There is a potential possibility that the Kafka write could fail, for whatever reason, and you have returned back to the user so you'll miss certain analytics. Now, in this kind of a system, because payment and all is not involved, it's just some simple analytics, we should be okay if we are losing out on certain events. So, I think it's okay if we build it this way also. Worst case, we'll just lose out on some analytics, which is fine. But, can we improve it even further? So, the problem with this approach is - each time we write to Kafka, it involves some I/O (input/ output). There's a CPU involved, there's a I/O involved, there's a network transfer involved.

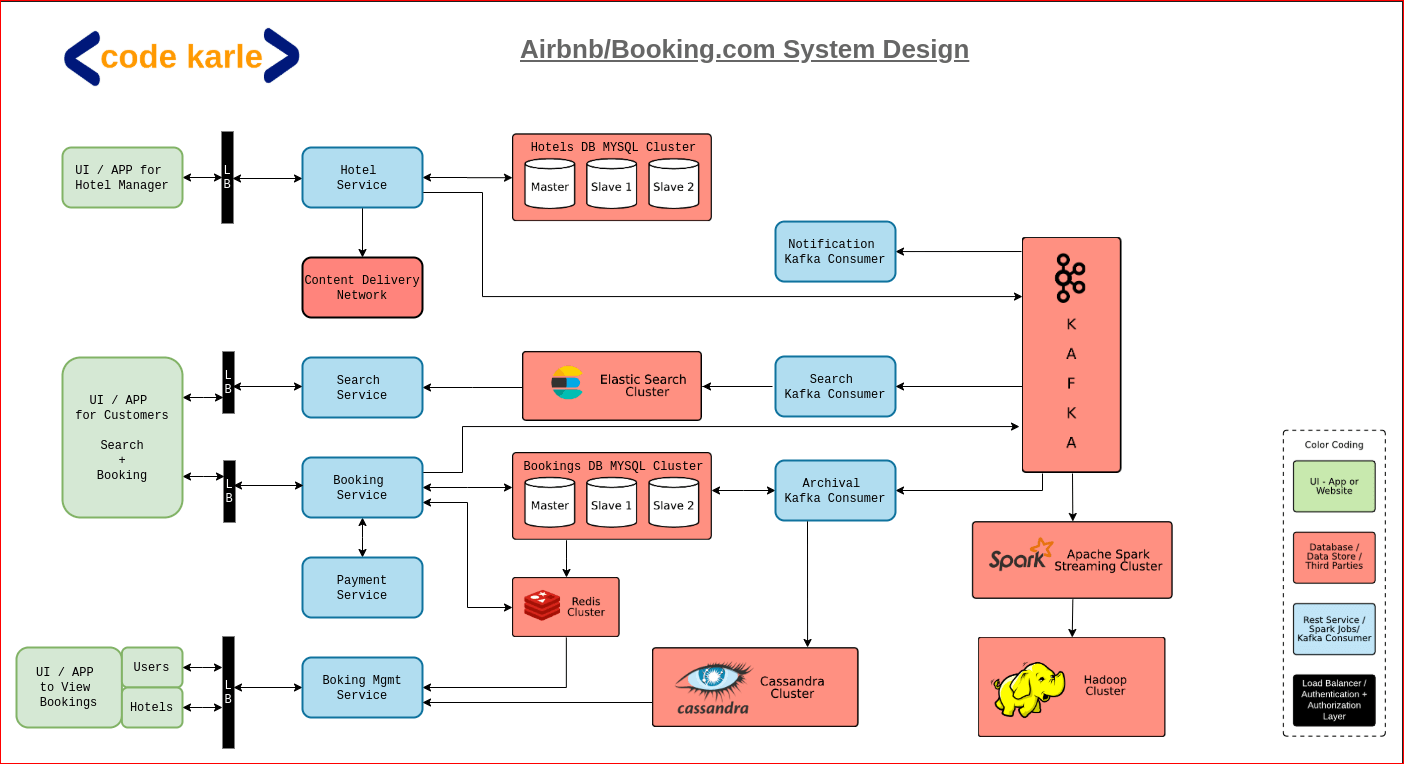
Can we avoid all of this i.e. doing all of this on each call? Probably! So, what we could do is - instead of writing into Kafka on each request, we could maybe aggregate that information locally in a machine. So, maybe you can have a queue or some kind of a data structure in which you are persisting each record that we got a request for this short URL with count 1. The request again comes in, you increment the count to count 2. Something of that sort you could implement. Or you could simply make it as a queue of all the requests. Whenever the size of the data structure crosses some threshold, or you could do it on a time basis also, saying every 10 seconds you'll flush that. So, in whatever condition you can flush the data into that data structure, into one call in Kafka. Basically you are reducing the I/O that you are doing with each request and doing it as a batch write. The benefit you get is - you can have now drive much more performance out of that single machine, thereby helping us in the non-functional requirements of high availability, low latency, all of that.

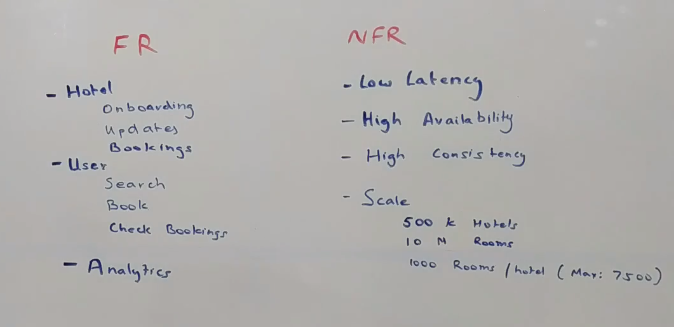
And secondly, it will improve the overall performance of the system. Again, the drawback here is - now you can lose out, not just one event but a lot of events. Based on your conversation with your interviewer, you can decide on - if that's okay or not. Personally, I would say it's a fairly okay thing if you lose out on 10, 20, 30 events. That's perfectly fine.

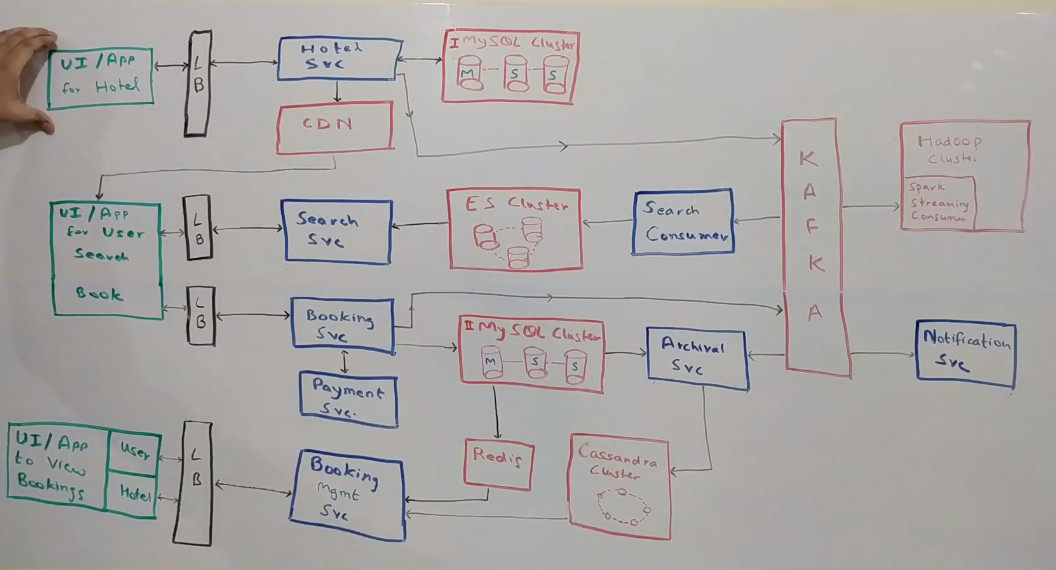
Now, we have put all the events into Kafka. Now what happens? So, there are a bunch of ways you can implement Analytics. One very simple approach is - you can dump all of this data into a Hadoop and then build some Hive queries and some kind of queries on top of it, which will generate you aggregate results. Alternatively, you could also have a Spark Steaming job running, which comes up every 10 minutes, let's say and takes all the data in last 10 minutes and does some aggregate analysis on top of it, saying which URL was it, how many times, which geographies people came in, how many times, something of that sort and dumps the aggregate information into a data store, which can then be used to power various kinds of analytics that user can see. So, you could implement it in either way you want, based on the conversation with your interviewer.

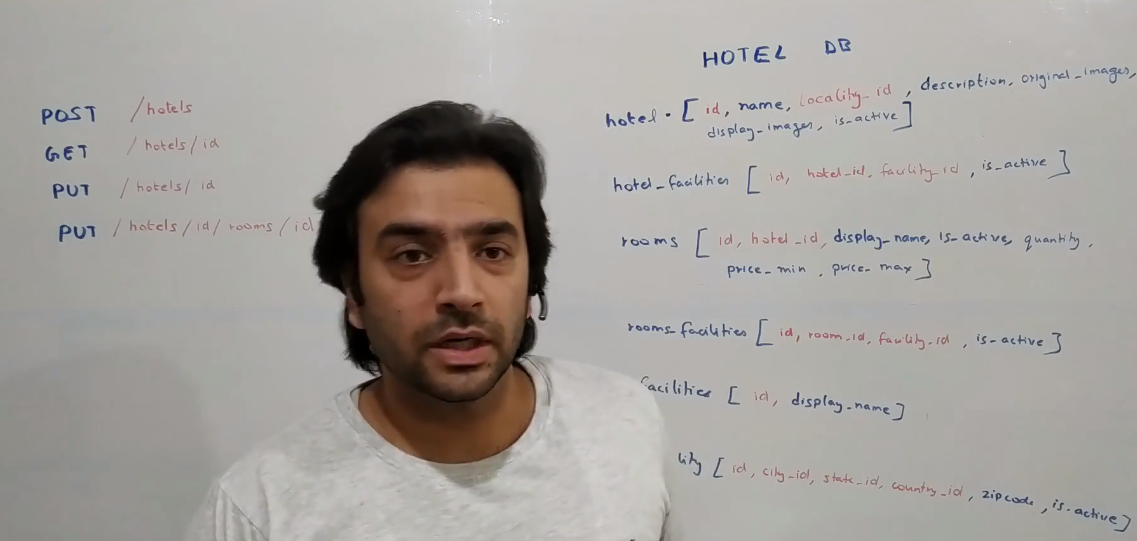
So yeah!! I think this should be it for a URL shortening system.

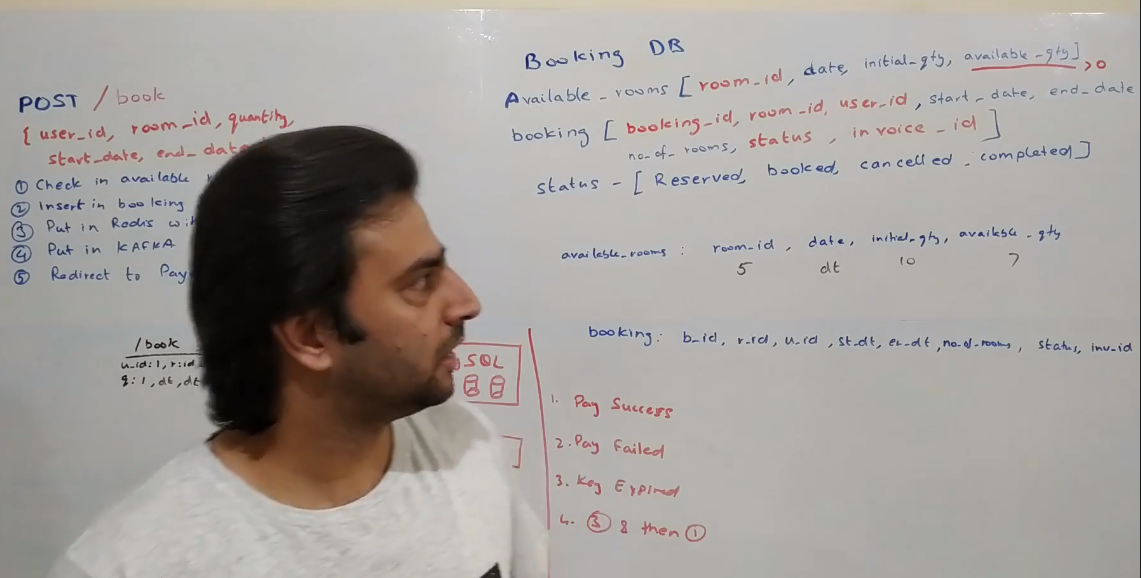
4. Airbnb / Booking.com System Design













so let's look at how do we design a hotel booking system something very similar to booking.com or airbnb but just one thing to call out

we will be looking at a very high level architecture of the whole system and not at a lower level class diagram and all of that in this video so before we jump into the problem let's first look at the functional requirements then we look at the non-functional requirements of what we want to achieve and then we look at the design so we have two major consumers of this application one is the hotel side of users and then there are the consumers who want to book the hotel so for the hotel managers we'll have these three major functionalities

1) they should be able to onboard onto our platform

2) they should be able to update their property so for example they might want to add a new room they might want to change the pricing they might want to add new images and stuff like that

3) then they should be able to see what all bookings are there and along with that also get some insight into the revenue numbers and all of that

From a user standpoint they should be able to search for a property in a particular location with a couple of search criteria so for example they might want to filter within a price range or some aspects of the property like a five-star property or a beach front property and stuff like that. Then they should be able to book that hotel and once they have booked they should be able to look at their bookings okay these are the major requirements now we should also design it in a way that we leave scope for some kind of analytics to be done so these are the functional side of things from a non-functional side of things we need this platform to run at a very low latency and it should give a very high availability and a very high consistency by high consistency I mean if you are booking a hotel or if a user is booking a hotel he should be able to see that hotel immediately now from a scale standpoint what kind of scale do we want so a quick google search tells me that there are roughly 500,000 hotels in the whole world at this point in time there are roughly 10-12 million rooms in all the hotels across the world at this point in time and roughly there... you can assume that there are thousand rooms in a particular hotel in general so there are some hotels who have which have more than 7000 rooms at this point in time but those are some edge cases we should be able to handle that. The reason I am talking about this thousand number is so let's say a hotel has thousand rooms now these rooms will be booked over a course of many days so there will never be a situation that there is just one room available and there are thousands of users who are wanting to book that. At max what will happen is that there is one room and there are two three users who are trying to book that and we will be able to use that assumption for our leverage at later point in time.

Now let's look at the overall design of the whole system and how the data flows within each component then we look individually into some of the components. So the whole business flow starts at this point which is basically a UI that we give out to the hotel managers through this you it could be either a website or a mobile app but through this UI they would come on onboard onto our platform and the same UI would be used by them to modify the property. So let's say they want to add a new image or they want to add a new room or if they want to make any modifications this is the UI that they talk to. Now this UI talks to a Load Balancer through which it talks to a hotel service. This is basically a service which manages the hotel part which is basically the onboarding and the management.

All right, now let's just say there's a spike in traffic so there could be multiple nodes of this hotel services that could be added here and so this becomes a horizontally scalable component Now hotel data in itself is a very much relational data plus earlier we talked about the number of photos that's not too many so it doesn't even have a scale problem so we'll be using a clustered MySQL here with one master and multiple slaves slaves can be added as and when required. let's say there's a huge spike in Read traffic, we can add more slaves but this data resides within MySQL database Now let's just say any image is added, so hotels can add images about the rooms about their whole building and all of that all those images would be stored into a CDN and the reference to the CDN which is basically a URL of the image would be stored in the database and that URL would be sent out to customers and whenever they want to render an image that would be looked up directly from the CDN.

Now what is a CDN? it's basically a geographically distributed data store which we will be using for sending out images throughout the whole world. So let's just say I'm connecting from India somebody's connecting from US they want to look up for an image of a particular hotel so I'll look up on the CDN server which is in India the other person will look up into the CDN server which is in US So this becomes the hotel life cycle management The next thing is basically let's just say each time a modification is happening to a hotel let's just say a new hotel comes in we want to bubble up this hotel to the users who are going to search for this right, now there are multiple ways in which we can send out this information to the search piece, right. I'll be using a Kafka here so each modification that is happening within hotel service will flow through a kafka cluster and there'll be multiple consumers that will be sitting on top of this cluster which will populate their data store for serving the search traffic and for other traffic as well, right. So one of the consumers will be the search consumer what happens is let's say a hotel gets a new room, for example. There will be a payload that is put into Kafka which has all the information that is required Now the search consumer pulls up the payload from Kafka and it stores into its own database and this database would be used to power the search on the website. Okay, now for search, I am using an elastic search.

Elasticsearch is basically a database that is built on Lucene platform. Similarly, instead of elasticsearch you could also use a Solr here. Both are kind of similar components ideally it would depend on what infrastructure is being used in your company you could use that right. But the idea of using elasticsearch is that I want this piece to be supporting fuzzy search now let's just say i am searching for a hotel in maldives or let's say user is searching for a hotel in maldives, the user might not know the correct spelling, right. If they type in a wrong word I don't want them to get no results. I would want this to be able to support a fuzzy search so I have to be able to handle all the typos and spelling mistakes and all of that plus i also want to give similar.. similarity kind of a thing there so that's the reason i'm using elastic search here. So all the data of each individual hotel, flows through the kafka via the search consumer into this elasticsearch cluster. Now on top of this elasticsearch sits the Search Service. Now again let's just say there's a spike in traffic I can increase the number of nodes in kafka cluster, I can increase the number of search consumers here and I can increase the number of nodes in elasticsearch cluster. So till now whatever we have talked about is again horizontally scalable, right. And again coming to Search Service this is the service which powers the search on the website now website is... i'm using a generic term sometimes I'll use a website sometimes I'll say UI but it's basically all modes of communication through which a user can come in. That could be an app, that could be a website right. So the user talks to through again a load balancer to the search service whenever they want to search for a particular hotel again they will give a date range and a location for example as a search criteria and along with that they could also provide some tags. Now those tags would be the properties of the hotels. So again going back to my previous example a five star property is a tag. A beachfront property is a tag. Now the search on elasticity would be happening on either of these tags and the ranges that are provided basically the date range, price range and all of that. okay, so this takes care of the search flow.

Now once the user has seen some of the results on the website they would want to book a hotel. The booking again happens through this UI. So I've made this UI saying that it's a search and book UI. Normally, it will be the same app or the same website through which they are searching and then booking right. Now a booking request again comes to this load balancer and talks to Booking Service. Booking Service essentially again sits on top of a MySQL database now these are two different MySQL clusters. I am purposefully not using a same cluster here although we could use the same cluster and have two different databases in that but because we are talking about a fairly large system that has like a good enough amount of scale I would want to keep different clusters so as to you know take care of the scaling separately of each other Now, whenever a booking happens, that booking gets stored into this MySQL we go over the exact flow of booking when we go over the details of implementation within the Booking Service, but essentially this stores the data into this MySQL and it talks to a Payment Service Normally what will happen... a booking request will come, it stores something, it will send the request for payment, once there's a success, it will mark the booking confirm. Now again, whenever a booking is happening, the data is flowing into the same kafka, right. Why? so let's just say there was just one room available in a hotel right and that room is now booked i want to make sure that this hotel is not available for search now in that same date range, because it's not available. So all of those information is again sent to the same kafka which is read by Search Consumer and then it takes care of even removing the hotels which are now completely booked. Now if you can see, there's something called an Archival Service here.

What I have done is, I am just storing the live data into MySQL. By live data, I mean the bookings that are done but have not been completed thereby making sure that this is having a scale which is low enough that MySQL can easily handle, and once the booking moves to a terminal state so let's say booking is cancelled or booking is completed it will move through the archival service to a Cassandra cluster. The reason I'm using a Cassandra here is so cassandra is a very good database which can handle a huge amount of reads and writes. It has a constraint that it needs a partition key on which all the queries should happen. So let's say if I want to search by a "booking\_id" my partition key has to be a "booking\_id" in that case I cannot do any kinds of queries on a Cassandra therefore I did not use a Cassandra as a source of truth database. Because on this database I need to do a large variety of queries.

We'll come to all of those when we go into the detail of Booking Service, but once it is archived we just need to do GETs on those. So therefore Cassandra makes a good enough sense over here. Now, once the booking is done, all of that is fine, but now we need to notify all the people right? So then comes the Notification Service. So let's say whenever a booking is made, or any changes are happening into a booking or it moves into a terminal state, there'll be a Notification Service that consumes events from this kafka and notifies the people, so for example on each booking, we need to notify the hotel right. Whenever a booking is cancelled by the hotel we need to notify the consumer or in fact on each booking we need to notify the consumer with an invoice right. So all of those is taken care by this Notification Service. Now coming back to the UI for hotels and users. So each time a booking is done or even without that a user might want to see their old bookings or a hotel might want to see all the bookings that they have. This is more of a read-only view for them, right? That will be powered by this Booking Management Service, which talks to now two data sources. It talks to the MySQL cluster for all the active bookings, which are to happen sometime in future and to the Cassandra cluster, for the bookings that have already happened right. Now i am adding a Redis on top of this MySQL to reduce the load on this MySQL, so Redis will act as my Cache and whenever I have a query so for example something like get bookings of a user so I can cache this result into this Redis. And it'll be a write-through cache, so whenever a new booking is coming in this will get updated all right.

Now this is the functional flow the bigger component here is how do we do the analytics on this so let's just say a business person wants to know how much revenue I'm making or how many bookings I'm having or what are my best performing hotels and stuff like that. So they need to do a lot of analytics Now mostly while designing the system we'll never always know what kind of analytics is required right so what I've done for that is I've used a Hadoop Cluster on which I'm pushing in all the events that are going into my kafka. Which is basically information about all my hotels, about all my bookings, about all the transactions that happen in my system. So there will be a Spark Streaming Consumer that runs somewhere that reads from this kafka and puts all the data into a Hadoop Cluster on which I can do Hive queries or any different kind of queries and build up a lot of reporting. So this is overall how the system looks like and how the data flows. Now let's go into the details of some of the components. Now let's look at what Hotel Service internally is. So it's not a very complicated service it is basically a CRUD Service which provides Create, Update, Read, Delete operations on the hotel data store. And it is the source of truth for hotel data.

Now, this is not an exhaustive list of neither the APIs nor the DB Schema that you see here there will be a lot more things, but this will give you a feel of how it should be.

So let's look at some of the APIs.

1) There'll be a POST API /hotels to create a hotel which will be part of their onboarding process.

2) There will be a GET API with an id GET /hotel/{hotel\_id} which will give back the information of the hotel which can be rendered on the screen and the hotel guy can see it.

3) There will be a PUT API PUT /hotel/id which will be used to update any information of a hotel.

4) Similarly there will be a PUT API PUT /hotel/{hotel\_id}/room/{room\_id} which would be used to update the room information or create new rooms and all of that.

Now this is not an exhaustive list there'll be a lot more APIs that you can add it as in when you know there's a requirement to add. Now let's look at how the DB schema might look like. So there are a couple of important tables now this is again not an exhaustive list of databases of the tables so there's one hotel table into this hotel DB but before that everything in red here is either a primary key or a foreign key. everything in blue is just a column now this hotel table contains your very standard things id, name, locality\_id which is a foreign key to locality table description, original\_images, display\_images and is\_active.

Now I have two columns for original and display images? so original\_images is basically the artifact that the people have uploaded display\_images could be a compressed version of that, that we've compressed, it could be a version that we have uploaded on the CDN, it could be something different than the original image but we still need to keep both of them so we have stored it here. is\_active is basically like a soft delete flag Then coming to rooms table. It has a room id obviously, a hotel\_id which references into this[hotel] table a display\_name which could just be a identifier to tell the customer on what kind of a room it is, is\_active again a soft delete flag, quantity basically tells how many such rooms are there in the hotel and a price\_min and a price\_max.

Now why do I have do we have two prizes? Remember the hadoop cluster that we had in the original design that we made. it has a lot of data about various kinds of things we might as well run a machine learning model onto it and do some supply demand analytics and then come up with the optimal price! right? let's say supply is low there's a lot of demand there are just a few rooms left... might as well increase the price! or let's say if there are too many rooms and very few customers might as well reduce the price. So this price\_min and price\_max could be the ranges which the hotel provides, wherein the price could be fluctuated by the system. A good starting point could be an average of both these prices right. Then there's a facilities table, which is basically a list of all the facilities that a hotel and a room can possibly have and these hotels\_facilities and room\_facilities are basically mapping tables which is a many-to-many relationship between a hotel\_id and a facility\_id. again is\_active flag everywhere is basically a soft delete flag. now again this is not a full list of tables there are a lot of information missing. I've skipped the auditing information, I've skipped the bookkeeping information like created\_on, updated\_time and all of that.

A lot of information missing but this will give you a fair enough idea and it will be a good starting point for you to come up with a DB schema for this. One more thing to note here that if you remember the original design that we had I did not keep a Redis cache on top of this MySQL database but I did keep a Redis cache on the other MySQL database which was for Booking DB Now why is that? We could have kept the Cache on top of this and all these GET APIs could have been a bit more faster right, but this is not coming in the critical path of any high throughput business interaction right so all the customers are not querying this database, neither this service, they are always querying the Search Service so if this service is a little bit slow that's okay but adding a Redis Cluster is a cost. So you need to do a trade-off analysis between what cost are you adding of an infrastructure and what benefit it adds to you if it is worth it you might as well go and add a Redis cluster here, but I don't think it is worth it and that's the reason I did not add it.

Now let's look at the internal functioning of the Booking Service. We'll first start off just walking through the DB Schema again it's not a full-fledged schema there are a lot of details missing like bookkeeping information like created\_time, updated\_time and all of that but let's focus on the meaty part here. So it has a table called available\_rooms which has a room\_id it has a date, it has an initial\_quantity that comes from the hotel service and it has a available\_quantity available\_quantity is basically the number of rooms that are remaining for that particular room\_id for that particular date. Now, it has a constraint saying it cannot go negative. Here is where the true power of MySQL we are utilizing and that's the reason why I chose to use MySQL here the other table here is a booking table it has a booking\_id which is the primary key here, which will be referenced across the whole system it has a room\_id, again comes from the room table it has a user\_id, a start\_date and an end\_date for a particular booking, number\_of\_rooms which is how many rooms the person has booked, status and an invoice\_id. looking at this design we can clearly understand that one booking cannot contain different room types you can have multiple rooms of the same room type but you cannot have like one deluxe room and one regular room in one booking. If you want that there'll be a small change required but i think that's a minor detail it can be taken care of easily. The important part here is the status column. it has these four values - reserved, booked cancelled and completed. Now canceled and completed are the terminal statuses here so the booking gets first created into reserved status. Then based on the payment success it can either move to book or cancel. And once the user stays in that, it moves to complete. Now you can add more statuses depending upon your conversation with your interviewer but these four statuses are the main ones that will help us achieve what we initially thought of.

Now, let's look at the API Signature so this will have one important API called a book API it will be a POST API which will take these five attributes. It will contain a user\_id it will contain a room\_id, it will contain the quantity. Now again if you want to make multiple rooms multiple quantity we'll have to change it a bit to have an array but let's stick to this for now it'll have a start\_date and it'll have a end\_date. The price will come from somewhere else let's assume for now. It'll actually come from the data store which contains the price for the room at this point in time we don't want to generally take the price from the user because then the request can be tampered with and that's not really a good design okay now let's do a quick revision of the design because i skipped some important details in the earlier larger diagram and we'll go over that now. So the way Booking Service actually works is when it gets a request to do a booking it first of all queries this table and the available\_rooms table and check whether or not I have that many number of rooms remaining or not. So if there are no rooms left for that particular room\_id for that particular date, there's no point of proceeding so we can error out from there. But in case that's a success and we have rooms then we actually go ahead with the blocking of the room saying that now I'll block it temporarily and if the payment is success I'll actually book the room. I'll do a quick dry run of what actually happens.

Assuming this is the request that came in user\_id: 1 | room\_id: 5 | quantity: 1 for some date "dt" to "dt +1". The room\_id: 5 on that particular date "dt" has 7 available rooms. So our first check is a success that we have enough rooms. So what essentially will happen is there'll be a row created in this table with a booking\_id: (some\_uuid) | room\_id: 5 user\_id: 1 | start\_date: dt whatever that is, end\_date would be "dt + 1" whatever that is, number\_of\_rooms in the request is quantity:1, and status would be at this point in time, RESERVED. invoice\_id at this point in time would be NULL because there is no invoice created till now Now, we have a record, along with that we also decrement the quantity here now here again we are utilizing a very important feature of MySQL which is part of the ACID property and transactions. So we are creating a record here[booking table] and we are reducing the quantity here[available\_rooms] to 6. what essentially we are trying to do is basically bounding this as part of one transaction so let's say there was just one room left and two three requests came in only one transaction would be successful to do both these things. Basically to insert this record and reduce the quantity because we have this constraint sitting over here which says that quantity cannot be negative. okay so only one of the transaction will be success, and only one of the rooms will be booked and no two users will be redirected to payment. That being taken care of what is the next step so i have written down the steps here if you want to actually look at so what we have gone through till now is step number one and step number two okay we've inserted in booking and reduces reduce the quantity in available\_rooms our step number three is something that I did not cover as part of the larger design review because it was getting too much cluttered.

Now we cannot keep this room reserved for an infinite amount of time. What we can say is if the payment is success in next five minutes, well and good, if not then we'll assume that the payment will not go through and will unblock the room so that somebody else could book it, okay. So there are multiple ways to implement that what I choose to implement here is something using the TTL(Time To Live) of Redis. So because we anyway are using a Redis we can utilize the same cluster of Redis for this use case as well. So what we'll do is we'll put the key in Redis saying some booking\_id expires at some timestamp Now the time stamp could be a configurable number, it could be a fixed timestamp across the board, it could be a country specific timestamp, for India have an expiry time of five minutes, for US have expiry time of four minutes, something of that sort but whatever that time is, we'll insert that into redis. Now what redis does is, it has something called callbacks so one of the later versions of Redis has introduced this concept called callbacks so whenever a key is getting expired you'll get a notification, okay. And you can do whatever you need to do at that point in time, right. So, if you get a Success notification from payment, well and good. Success notification means the payment has gone through, then you will mark the booking as BOOKED but before that if you get a callback from Redis saying that the key has expired and you've not got the success from payment you will say that the booking is CANCELLED. Alternatively you could also get a failure from payment saying for whatever reason the payment didn't go through and you got a failure response from the Payment Service, in that again you can say CANCELLED.

Now if you want a bifurcation of the varieties of CANCELLED, you can maybe make multiple statuses like cancel because of invoicing / cancel because of payment/ cancel because of expiry, whatever right or you could maybe add a status\_reason column or something of that sort but that's a very minor detail we'll skip that for now, okay. So let's go over what all possibilities are there in this and how each of them behaves, okay. So, first very simple thing is - what happens when payment is a Sucess? so in case payment is a success everything remains the same just the status becomes BOOKED. okay, in that case, we do get some invoice\_id as well. So basically we'll get an invoice\_id from Payment Service whenever you know a booking is getting success and we'll just update the invoice\_id there and then the regular kafka events would also be sent saying the booking is now complete and here's the kafka event for that in case somebody wants to do something on that. What happens when payment fails? Now in this we just have these four statuses so the booking status will become CANCELLED. okay there would be no invoice\_id in this case Why? because if the payment did not go through there obviously is not an invoice that is generated. And everything else remains the same but, if the payment did not go through, we need to revert the available\_quantity again. so available\_quantity in that case would become seven.

Now let's say your key expired so basically let's say the user was redirected to payment screen and there was no response from payment service for whatever reason what happens then if we get a call back from Redis and based on that call back we can say that okay now the payment has not gone through we will follow the same process as payment failure we will mark this CANCELLED okay we will mark this CANCELLED and we'll increment the quantity in available\_quantity so that the room is now available for somebody else to use. Again in that scenario there is no invoice generated. But this you do only if the status is RESERVED. Why? - coming to the next case, what happens if both (3) and (1) happen. What happens if you get a key expiry event and a payment is also success. so there are two conditions if the payment has already been successful and the booking has already been moved to BOOKED status, after that if you get this key expired event then you don't do anything because that is any way bound to happen right? but what if it happens the other way around what if key expired first you move the booking to CANCELLED state but then you get a notification saying payment is success Now there are multiple directions in which you could take it based on your conversation with your interviewer and the non-functional requirements and in fact even the functional requirements for that matter you could do two three things. You could now either revert the payment saying for whatever reason we were not able to book the room so here's your payment back. Alternatively you could do something even more smarter. You could say that now I have anyway got the payment from the user, I can check if there are rooms available and I'll book them, right? Now this could be done based on what the requirement is and you could talk to your interviewer and implement it either ways.

All good so far but there are a couple of caveats here. The TTL that you have talked about it is not a very precise measure so let's just say that a key was supposed to expire at 10:00 okay, you will probably never ever get a call back at this point in time it will always have some delay. Now in this case it doesn't matter too much instead of at 10:00 if you get it at 10:01 it's possibly okay also. So it's not too big of a problem and the reason for that is because of the way expires are implemented in Redis I'll not go too much into detail of that but there's a background process that runs in Redis for keys that are not accessed and whenever that process gets to access a particular key is when it will expire it. So it is not necessary that it will acquire it at exactly the same time. But let's say if you wanted it to be totally precise then you could possibly tweak the implementation a bit and do a slightly different way so instead of doing a TTL based approach you could in fact implement a queue with it within Redis and have a poller that kind of queries Redis, the topmost node of the queue every one second and whichever one it it finds has expired then you could kind of delete that but that's not that's obviously much better but that comes at a cost so you'll have to build a kind of a polling mechanism so that's additional development effort and then it will be continuously bombarding Redis every one second so there's a lot of CPU being utilized on both the sides on the cron side, and on the Redis side so possibly you'll have to add more nodes into the Redis cluster and also on the side where cron is being developed So now that's a tradeoff. Do you want to be notified absolutely immediately when the keys are supposed to be expired and at the cost of additional hardware that trade-off you can again make with the conversation with your interviewer. But otherwise all of this being said I would still go with a TTL based approach because in this particular example it doesn't really matter so much.

Now a couple of optimizations you could do. So, let's just say payment is success. You know that key will expire after some time, for sure, right? because it's there in Redis. You don't need to keep that key there you know the payment is success you can evict the key right, even if the for the payment failure case you know that payment has failed it will expire after five minutes might as well delete the key then and there, right. So these are certain optimizations that you could do over this implementation to make it even more better. But on and off this is how the booking flow works. Now again reiterating we have used a couple of important features of MySQL and that's what is helping us to make the code on application side much more smoother had we used some other database which doesn't provide for example if you were using Cassandra here we would not have had access to the transactions and the constraints and all of that you would have to implement it on application side. That's additional effort on our side to make sure things are consistent. in this case I would rather leave it to MySQL to implement all of those things. Now coming back to the same architecture again i just want to call out that all of the components that you see here are individually horizontally scalable so let's just say there's a traffic spike happening on one of the components we could increase the number of nodes in that particular service maybe that particular database and then that should work just fine. As far as kafka and hadoop cluster are concerned we could add more nodes into that as well and they should also scale to a much larger scale than what we need. C

ool, so now let's look at what kind of alternates that we could have used instead of this particular design choice. So first of all why MySQL, we could use any other relational database here. We could use a Postgres we could use a SQL Server, anything which provides ACID guarantees should be fairly fine here. As far as Redis is concerned we could use a memcache or any other cache instead of Redis and that should also be good. Cassandra, I would still stick to Cassandra because that is exactly what we need here now technically in place of Cassandra we could also use a HBase here that would also work fine but it has a lot of operational overhead in terms of deployment and maintaining it over time so that's the reason I would prefer Cassandra over HBase or any other similar system The way cassandra works is every data in cassandra is you know sharded across a partition key so each query has to happen on a partition key now the queries that we are doing are just of two varieties.

1) Get bookings by hotel or

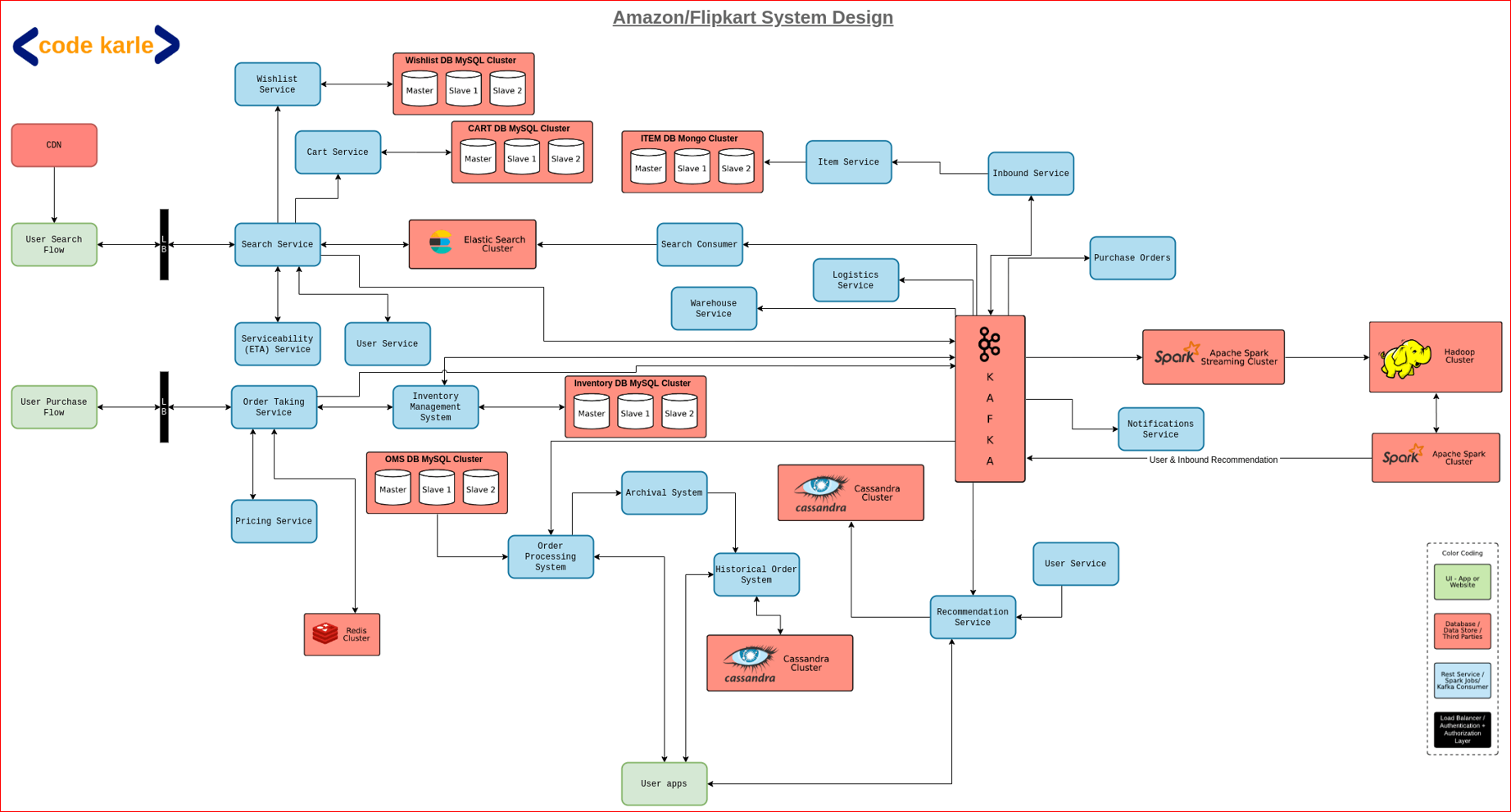
2) Get bookings by user. There is no the third variety so we basically have two kinds of data which is distributed by two different partition keys on which the queries are happening. So this would be kind of a very good choice here. In place of Kafka, we could have used an Active MQ or a Rabbit MQ or any other queueing mechanism there's an amazon queue(SQS) also we could have used that but i think kafka scales much better than most of them so i think it's a fairly good choice here. Other than that in general we definitely need to monitor how are our CPUs and Memory is behaving. So if I have a CPU spike at certain points in time that is something we need to kind of look at so across the whole infrastructure we need to keep an eye on how my CPU usage percentage is, how my memory usage percentage is, how my disk usage for Redis is, how my disk usage for elastic search is all of these things are what we need to monitor now monitoring could be done through a grafana kind of a tool on which i can set up alert. So if the let's say a particular metric has some threshold the moment I cross that threshold or with certain conditions I could send out an alert and the team could get notified that something is potentially wrong and they need to look at that. this will help us to make sure that we in the end achieve our NFRs that we talked about of latency and high availability. Because let's just say something goes wrong let's just say memory is you know utilize more than what we expected eventually it will lead to some machines going down and eventually it will lead to us having a lower availability that than what we expected so yeah these are the things that we need to monitor and alert on.

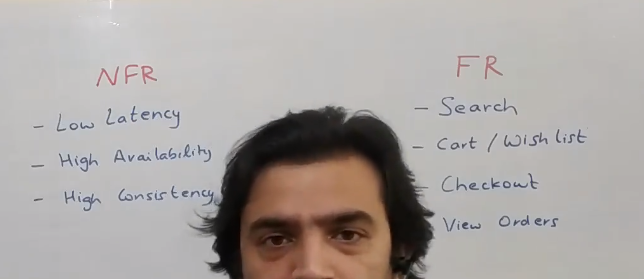
Now in the next section let's look at how this whole thing would be spread across geographies, so for example let's say there's an earthquake in one of the data centers and everything just goes away out of the blue what do we do? So let's look at that next So let's say we have these four data centers data center 1, data center 2 data center 3, and data center 4 which are located in different geographical regions across the globe, okay. Now we want to create a topology in a way that we do get low latency and high availability okay so one very simple approach that we could do is say that DC 1 is our primary and all the three DC's are our secondary data centers and data is replicated to all the three data centers in near real time okay, so that's okay it's good enough. but it's not very good to be honest because we are just using 25 percent of our capacity as primary which is active and rest three data centers are sitting idle and not really doing anything. So let's try to improvise what we could instead do is divide the data centers and thus the globe into two parts.

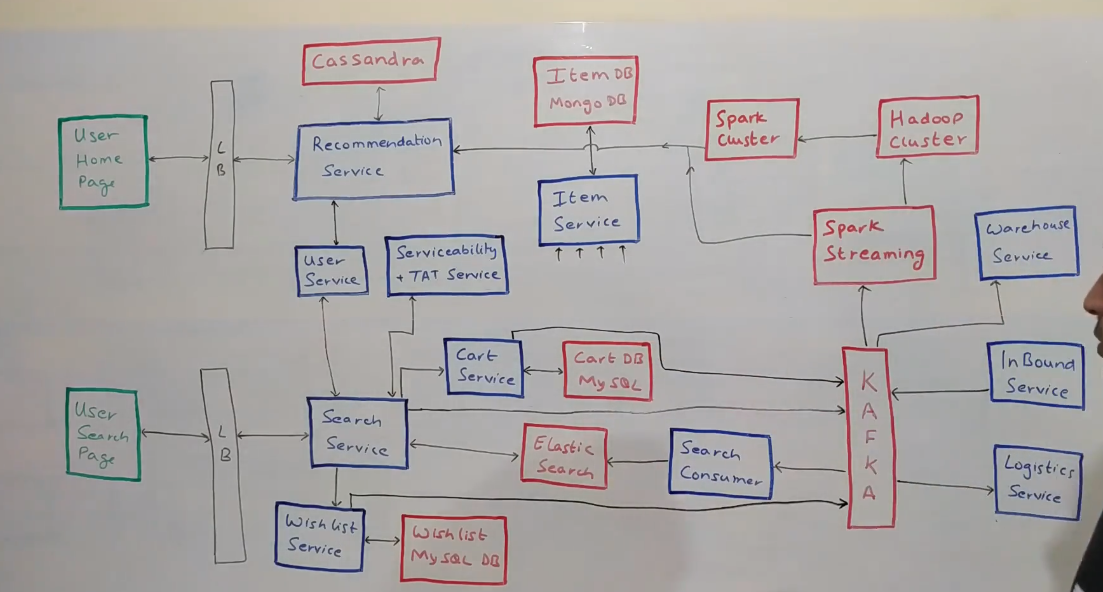
What we could say is this is region one and this is region two okay now the countries or people accessing our services who are closer to this region(R1) will connect to this region(R1) and the people who are closer to that region(R2) will connect to that region(R2). Now how are we able to do that so the data in a hotel management system is fairly specific to a geography so all the hotels in let's say India can be you know separated from all the hotels in US. Similarly all the rooms all the bookings are now specific to hotels and thus specific to geography so we could kind of bifurcate the data as per geography right which gives us the leverage to divide the system into two halves right now what will happen here now let's just say DC1 is the primary in this region(R1) and DC3 is the primary in R2 okay now if DC1 goes down all the data in DC1 is getting replicated to DC2 in near real time so if that goes down DC2 can become active and all the clients who are connecting to DC1 and how will they connect so there will be bunch of clients who are connecting via some DNS to DC1 right if this goes down DNS can flip and connect to DC2 if this link is broken right similar thing can happen on this side so this way what we have is basically dividing our infrastructure into two halves thereby clients who are closer to this region are connecting to the servers that are closer to them thus giving them lower latency right.

Now we could go even one step further we could say that we'll divide the region into four parts and we could do we could go as much as deep we want into this to increase the latency basically to reduce the latency and increase the availability but i think for all practical purposes at least for a Hotel Booking System this R1 R2 thing is more than sufficient to give us a good enough latency and a very high availability. So I think yeah that should be it for a Hotel Booking System.

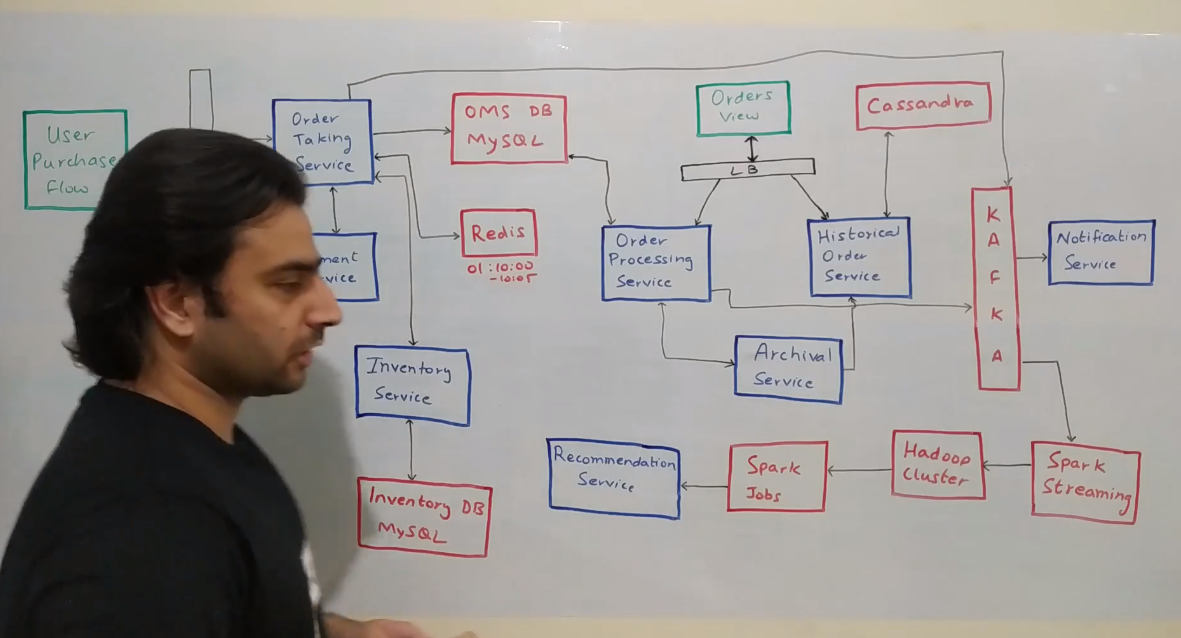
5. Amazon/Flipkart System Design





UI on User Services, User home page & Search page

Inbound services where suppliers send the product information



User purchase flow, order management service (above)

3 - Amazon/FLipkart

=====================

Hi everyone! Welcome to CodeKarle. In this video, we'll be looking at another very common System Design Interview problem, which is to design an ecommerce application, something very similar to Amazon or Flipkart or anything of that sort. Let's look at some functional and non-functional requirements(NFRs) that this platform should support. So the very first thing is that people should be able to search for whatever product they want to buy and along with searching we should also be able to tell them whether we can deliver it to them or not. So let's just say a particular user is in a very remote location where we cannot deliver a particular product then we should clearly call it out right on the search page. Why? - because let's say a user has seen hundred products and then they go to checkout flow add it into cart and then if we tell them that we can't deliver then that's a very bad user experience. So at the page of search itself we should be able to tell them that we cannot deliver or if you are delivering then by when should we be able to deliver it to you.

The next thing is there should be a concept of a cart or a wishlist or something of that sort so that people can basically add a particular item into a cart. The next thing is people should be able to check out which is basically making a payment and then completing the order. will not look at the how the payment exactly works like the integrating with Payment Gateways and all of that but we'll broadly look at how the flow overall works.

The next thing is people should be able to view all of their past historical orders as well. The ones that have been delivered , the ones that have not been delivered, everything From the non-functional side the system should have a very low latency.

Why? - because it will be a bad user experience if it's slow. It should be highly available and it should be highly consistent. Now high availability,

high consistency and low latency all three together sounds a bit too much so let's break it down. Not everything needs to have all these three So some of the products which are mainly dealing with the payment and inventory counting, they need to be highly consistent at the cost of availability. Certain components, like search and all, they need to be highly available, maybe at the cost of consistent at times.

And overall most of the user facing components should have a low latency.

Now let's start with the overall architecture of the whole system. We will do it in two parts

First, we look at the home screen and the search screen and then we look at the whole checkout flows. And before we get to the real thing let's look at a bit of convention to start with. Things in green are basically the user interfaces. It could be browsers, mobile applications, anything.

These two are basically not just load balancers but also reverse proxies and an authorization and authentication layer in between that will authenticate all the requests that are coming from outside world into our system. All the things in blue that you see here are basically the services that we have built. It could be

Kafka consumers, could be Spark jobs could be anything. And the things in red are basically the databases or any clusters could be Kafka Cluster/Hadoop cluster or any public facing thing that we've used. Now let's look at the user interface what all we'll have.

So we mainly have two user interfaces. One is the home screen which would be the first screen that a user comes to it would by default have some recommendations based on some past record of that user, or in case of new user some general recommendations. In case of search page it would just be a text box where in user head put in a text and will give them the search results.

Data FLow

--------------

With that let's look at how the data flow begins. So a company like Amazon would have various suppliers that they would have integrated with now these suppliers would be managed by various services on the supplier front okay. I am abstracting out all of them as something called as an Inbound Service. What that does is basically it talks to various supplier system and get all of the data Now let's say new item has come in. Or a supplier is basically onboarding a new item. That information comes through a lot of services and through Inbound Service into a Kafka. That is basically a supplier word coming into the whole search and user side of things. Now there are multiple consumers on that Kafka, which will process that particular piece of information to flow into the user world.

Item Service

=============

Let's look at what that is. So the very first thing that will happen is basically something called as an Item Service. So Item Service has a lot of people talking to it and if I start drawing all the lines, it become too cluttered so I've not on any connection to item service but item service will be one of the important things that listen to this kafka topic and what it does is it basically on board a new item. Now it also is basically the source of truth for all items in the ecosystem. So it will provide various APIs to get an item by item ID, add a new item, remove an item, update details of a particular item and it will also have an important API to bulk GET a lot of items. So get a GET API with a lot of item IDs and it is in response it gives details about all of those items. Now Item Service sits on top of a

MongoDB. Why do we need a Mongo here? - So item information is fundamentally very non structured. Different item types will have different attributes and all of that needs to be stored in a queriable format. Now if we try to model that in a structured form into MySQL kind of database that will not be an optimimal choice.

So that's why we've used a Mongo. To take an example, let's say if you look if you look at attributes of some products like a shirt, it would have a size attribute and it would probably have a color attribute right. Like a large size t-shirt of red color. If you look at something like a television, it will have a screen size, saying a 55 inch TV with the led display kind of a thing. There could be other things like a bread, which could have a type and weight. Like 400 grams of bread of which is of wheat bread or a sweet bread or something of that sort. So, it's a fairly non-structured data and that's the reason I use a Mongo database here.

Search Consumer

=============

Now, coming to other consumers that will be sitting on that Kafka. So on this Kafka, there is something called as a Search Consumer. What this does is basically whenever a new item comes in, this search consumer is responsible for making sure that item is now available for the users to query on. So what it does is all the items that are coming in it basically reads through all of those items. It puts that in a format that the search system understands and it stores it into its database.

Now, search uses a database called Elastic search, which is again a NoSQL database, which is very efficient at doing text-based queries. Now this whole search will happen either on the product name or product description and maybe it will have some filters on the product attributes. Those kind of things is something that elastic search is very good at. Plus in some cases you might also want to do a fuzzy search, which is also supported very well by Elastic Search. Now on top of this Elastic Search, there is something called as a Search Service. This Search Service is basically an interface that talks to the front end or any other component that wants to search anything within the ecosystem. It provides various kinds of APIs to filter products, to search by a particular string or anything of that sort. And the contract between Search Service and Search Consumer is fixed so both of them understand what is the type of data that is stored within Elastic Search. That is the reason consumer is able to write it and the Search Service is able to search for it.

Serviceability and TAT Service

==============================

Now, once .. a user wants to search something, there are two main things around it. One is basically trying to figure out the right set of items that needs to be displayed but there is also an important aspect that we should not show items that we cannot deliver to the customers. So for example, if a customer is staying in a very remote location and if we are not able to deliver big sized items like refrigerator to that pin code, we should not really show that search result to the user as well because otherwise it's a bad experience that he will see the result but you will not be able to order it, right. So search service talks to something called as a Serviceability and TAT(Turn Around Time) Service. This basically does a lot of things first of all it tries to figure out that, where exactly the product is. In what all warehouses. Now given one of those Warehouse or some of those warehouses, it tries to see - do I have a way to deliver products from this warehouse to the customer's pincode? and if I have, then what kind of products can I carry on that route. Now certain routes can carry all kinds of products but certain routes might not be able to carry things like big products or other kind of product, right. So all of those filtering stays within the Serviceability and TAT Service. Now it also does one more thing that it tells you in how much time I will be able to deliver, okay. So it would be like in number of hours or days or anything of that sort it can say that probably I will be able to deliver it in 12 hours, or 24 hours or something of that sort. Now if serviceability tells that I cannot deliver, search will simply filter out those results and ignore that and return the rest of the remaining things. Now Search might talk to User Service.

There's something called as a User Service here. We'll get into that later but the User Service is basically a service that is the source of true so for the user data and it(Search) can query that(User Service) to fetch some attributes of user probably a default address or something of that sort, which can be used basically

which can be passed as an argument to Serviceability Service to check whether I can deliver it or not, okay. Now Search Service returns the response to the user which can be rendered and the people can see whatever you know they want to see. Now each time a Search happens, an event is basically put into Kafka.

The reason behind this is whenever somebody is searching for something they are basically telling you an intent to buy a kind of product, right? That is a very good source for building a Recommendation. We'll look at how Recommendation can be build later on, but this is an input into the recommendation engine and we will be using a Kafka to pass it along. So each search query goes into Kafka, saying this user ID searched for this particular product. Now from the Search Screen user could be able to wishlist a particular product or add it to cart and buy it, right? All of those could be done using the Wishlist Service and Cart Service.

Wishlist Service is the repository of all the wish lists in the ecosystem and the cart services by repository of all the carts in the ecosystem. Carts are basically a shopping bag, which when people put product into it and then checkout. Now both these services are built in exactly the same way(almost). They

provide APIs to add a product into user's cart or wishlist. Get a user's Cart or Wishlist. Or delete a particular item from that. And they would have a very similar data model and they are both sitting on their own MySQL databases. Now from a hardware standpoint I'd like to keep these two as separate Hardwares. Just in case, for example Wishlist size becomes too big and it needs to scale out so we can scale this particular cluster accordingly. But otherwise functionally both of them are totally same.

Recommendation Service & Spark Streaming

========================================

Now each time, a person is putting a product into Wishlist, they are again giving you signal. Each time they are adding something into their cart they are again giving a signal, about their preferences, things that like that they want to buy and all of that, right? All of those could again be put

into Kafka for a very similar kind of analytics. Now let's look at what those analytics would be? - So from this Kafka there would be something like a Spark Streaming Consumer. One of the very first things that it does is - kind of come up with some kind of reports on what products people are

buying right now. Those would be things like coming up with the report saying what was the most bought item in the last 30 minutes, or what was the most Wishlisted listed item in last 30 minutes, or in Electronics category what which product is the most sought-after product. So all of those would be inferred by

this Spark Streaming. Other than that it also puts all the data to Hadoop saying this user has liked this product, this user has searched for this product anything that happens, right. On top of it, we could run various ML Algorithms. I've talked about those in detail in another video(Netflix Video), but the idea is - given a user and a certain kinds of products that they like, we could be able to figure of two kinds of information. One is what other products this user might like okay and the other thing is how similar is this user to other users and based on products that other users have already purchased we would recommend a

particular product to the user. So all of those is calculated by this Spark Cluster on top of which we can run various ML jobs to come up with this data. Once we calculate those recommendations, this Spark Cluster basically talks to something called as a Recommendation Service. Which is basically the repository of all the recommendations, and it has various kinds of recommendations. One is given a user ID, if you store general recommendations saying what are the most recommended products for this user. And it will also store the same information for each category. Saying for electronics for this user these are the recommendations. For the food kind of a thing, for this user, these are the recommendations. So when a person is actually on home page they will first see all in just general recommendations and if they navigate into a particular category they'll see the specific recommendations we are specific to the category.

User Service

===========

Now we have skipped a couple of components. User Service is a very straightforward service which basically you can look at any other of my videos that have talked about it in detail(Airbnb Video), but it's a repository of all the users. It provides various API is to Get details of a user, Update details of user and all

of that. It'll sit on top of a MySQL database and a Redis Cache on top of it. Now let's say Search Service wants to get details of a user the way it will work is, it will first query Redis to get the details of the user. If the user is present, it will return from there. But if the user is not present in Redis it will then query the MySQL Database (one of the slaves of that MYSQL cluster), get the user information, store it in Redis, and return it back. So that's how User Services will work.

Logistic Service and one is Warehouse service

============================================

Now there are some other components that are here. One is Logistic Service and one is Warehouse service. Normally these two components come in once the order is placed. But in this scenario this Serviceability Service might query either of these two services not a runtime, but before catching the information to fetch

various attributes. So for example it might query this Warehouse Service to get a repository of all the items that are in the warehouse or it might query Logistic Service to get details of all the pin codes that are existing or maybe details about or the Courier Partners that work in a particular locality and all of with

that all of the information this Serviceability Service will basically create a graph kind of a thing saying what is the shortest path to go from point A to point B and in how much time can I get there. Now we have not covered this service in detail. I have also made another video on implementation of Google Maps. This is very similar to that. So if you want to get into details you can look at that(Google Maps Video) but just to just to call it out this doesn't really do any calculation at runtime. You store all the information in a cache and whenever anybody queries, it will query the cache and return the results from the cache itself, and no runtime

calculation because those kind of calculations are fairly slow. And it will pre-calculate all possible requests that can come to it. Basically if there are N Pincodes and M Warehouses, it will do a M x N and calculate all possible combinations of requests that can come to the service, and store it in a cache.

Now let's look at what happens when a User tries to place an order. So the user interaction, to place an order is represented as this User Burchase Flow. This could be access to apps, mobile apps or web browsers or anything. Now whenever User says that I am ready to place an order take me to the payment screen

which is the last piece in the app flow then basically the request goes to something called as the Order Taking Service. Think of Order Taking Service as a part of Order Management System which takes the order.

Order Management System --> Inventory Service

===============================================

Now Order Management System sits on top of a MySQL database. Why MYSQL? - because if you look at order and table form for order, it will have a lot of tables, some with order information, some with customer information, some with item information and there are a lot of updates that happen on to the Order Database, right, for each order. Now we need to make sure that those updates are atomic and they can form a transaction so as to make sure that there are no partial updates happening. And because MYSQL provides that out of the box we would leverage that here. So that's why MySQL.

Now whenever this gets called the very first thing that happens is a record gets created for an order, an order ID gets generated. Now the next thing that we do is we put an entry into this Redis saying, this order ID was created at some point in time and this record expires at some point. Why is it being used, we'll come to. But think of it as an entry that goes in something like - Order id: "O1", gets placed and 10:00 let's say. it expires at 10:05. Something of that sort. Now the record that goes into this MySQL Database has an initial status. Think of it as a Status of: "CREATED". So basically the record that goes into this table is something like - order O1, was created at 10:00 with status: "CREATED". You can have different set of names also that's fine. The third thing that happens is we basically call Inventory Service. The idea of calling Invented Service is we want to block the inventory. So let's say if there were five units of a television that a user wanted to purchase, we will reduce the inventory count at this point in time, and then send the user to payment. Why do we do that? - Let's just say there was just one TV and three users want to come in at the same time. Now we want to make sure only one of them buys the TV and for two of them it should show that it is out of stock, right? We can easily enforce that, through this Inventory Service, by having a constraint on to the table which has the 'count' column saying that 'count' cannot go negative. Now each time you basically want to place that order for a TV, only one of the.. if the count is one, only one of the users will be able to do that transaction where the count reduces. For the other two, it'll be a Constraint Violation Exception because the count cannot be negative. So only one of them will go through with the inventory, right! Now once the inventory is updated then the user is taken to something called as a Payment Service.

Payment Service

================

Think of it as an abstraction over the whole payment flow where in this service talks to your Payment Gateways and takes care of all of the interaction with the payment. We will not go into details of how that works, but the interaction with the Payment Service can lead to two-three outcomes. One of them is that payment service could come back saying the payment was successful. Or it could say that the payment failed due to some reason, could be a lot of reasons. Or the person could just close the browser window after going to the payment gateway and not come back in which case payment service would not give a response. So these are there are

these three possibilities. Now let's see what can happen, okay. So one of the very first thing that can happen is let's just say at 10:01, we got a confirmation from Payment saying payment was successful. In that case it'll.. the order would get confirmed, right. So let's say we call it and that status is now "PLACED", or any other name for that matter. So now once the payment has been successful, we update the status of the order over here in the database saying that order is PLACED. Now the work is done. At that point in time an event would be send in to Kafka, staying an order has been placed, with so-and-so details, okay! Another option - it could(Payment Service) could come back saying that.. the payment failed. Let's say there was no money in the account or whatever happened. Now once the payment has failed, we need to do a lot of things. First of all, we need to cancel the order, because the payment has not happened. So let's say the other possible scenario is the order could be CANCELLED, right? Now.. if the payment has failed we need to increment the inventory count again. so we'll basically again call

Inventory Service and saying that - you decremented the quantity for this particular order ID, now let's increment it back. So that's a rollback transaction kind of a thing, that is happening here, okay. So that is one more thing we'll do. And also we'll have a Reconciliation Service that sits on top of this whole system which reconciles basically checks every once in a while that if there were ten orders overall I do have the correct inventory count at the end, just to handle cases with due to some reason we missed update of inventory count or the write fail. So all of those scenarios can be handled by the Reconciliation flow.

Now there is one more scenario that can happen. Payment Service could not at all come back, right? The user close the browser window, the flow ends there, payment service doesn't respond back to us. Now what do we do? We can't keep the inventive blocked. So let's say there was just one television available this user

has gone to the payment screen, closed the window and gone away. Now we still have that one television physically available in our warehouse but the database says it's not available. We need to bring it back, right. So for that the Redis comes in. So at 10:05 what will happen is Redis this record will get expired, right. Redis will have a will have a expiry. We'll have a expiry call back on top of Redis which will basically be invoked saying that this particular report whatever you insert inserted, it has got expired. Now do whatever you want. At that point in time Order Taking Service will catch that event and say that now this particular record has expired, I'll basically follow the same flow that was followed for payment cancellation basically I'll time out the payment and mark it CANCELLED, okay. So at that point in time again at 10:05, this order would be moved to CANCELLED State in the Database and the inventory would get updated back again, right. So everything is good but there are a couple of scenarios here. So there could be potential risk conditions. What happens if your payment success event and your expiry happens at the same time. What do you do then? So there are two three scenarios in which it would happen. The first scenario is that payment success comes first and then

expiry happens. So in that... that is a scenario that will always happen. So whenever a payment is success let's say at 10:01 payment got success anywhere at 10:05 this expiry would have happened, right? So this is something that is bound to happen for all the the orders. So one optimization we could do here is each time we get a payment success or payment failure event, we can delete the entry form Redis, so as to make sure it's expiry event does not come in. The other scenario is that expiry comes first and then payment success. Now whenever we would have got.. let's say expiry came at 10:05 and payment success happens at 10:07. Now whenever we get the expiry event we would have canceled the order, we would have decremented the inventory count. But now the pay ment has happened, that we got to know. There we could do 2-3 things. We could either refund the money back to the customers saying that for whatever reason we are not able to process, here's your money back. Alternatively, we could also say that now we have anyway got the money from the customer. We know what the person was able to... was trying to order. Due to a issue one our side possibly the order didn't go through. So we could create a new order mark this payment for that order, and then put that directly into PLACED status. So that is another thing that we could do. Assuming for all the orders as soon as we get a payment success and payment failure, we will delete these entries from Redis so to make sure there are no.. these conditions do not happen too frequently. That will help us to save the memory footprint as well, because the lesser amount of data we have in Redis, the lesser RAM it would consume, and it will be much more efficient okay. Now one thing I want to call out about this Redis expiry is - this is not very accurate. So let's say if it was the supposed to expire at 10:05, we might not get the call back exactly a 10:05. We might get a callback at 10:06, 10:07 or sometime after 10:05 and that is because of the way it is expiry works. So Redis doesn't expire all the records at that point in time. It basically it checks all the

records every once in a while for expiry and whenever it finds some of the docs/ some of the keys that are expired then it expires that. So it might happen that it'll expire after a few minutes. Now in this scenario it doesn't really matter so I think we should be okay with this kind of an approach but if you are

trying to do something which is much more mission-critical, then possibly you might have to do a different approach. But even you should talk about this to your interviewer, and if they say then you might also change the implementation a bit. You might implement a queue and keep polling that queue every

second or something of that sort. Okay. Now once the payment has happened or not happened or whatever happened, you would put all of those events into Kafka anyway. There is one more reason why you want to put two events. So let's say there was this one item one television left okay and somebody has purchased

that. Now you want to make sure that nobody else is able to see the television because anyway the count has become.. inventory count has become zero you cannot take an order, right. So you need to... basically make sure that it should be removed from search. Now remember there was a Search Consumer

that we looked at in the previous section. That Search Consumer would also take care of inventory kind of things. So as soon as an item goes out of stock it would basically remove that element from in the listing. Basically it will remove the documents pertaining to that particular set of items.

Archival , Historical Order service, Order Processing Service

=============================================================

Now there is one problem in this whole thing that we have decided/built. This MySQL can very quickly become a bottleneck. Way? - Because there are a lot of Orders. So for a company like Amazon they will probably have like millions of Orders in a day a couple of millions at-least. And that number could bloat up significantly

over an year. And normally you would have to keep Order Data for a couple of years for auditing purposes, right. So this Database is going massively big. we need to do something about it. So remember the whole use case of having this MySQL is to take care of atomicity, transaction, basically the acid properties for orders that are

being updated. But for the orders that are not being updated, we don't really need any of these functionalities, right. So what we can do is once you order reaches a terminal state, let's just say an order is DELIVERED or CANCELLED, we can basically move it to Cassandra. So there is something called as Archival

Service, which is a cron kind of a thing, which runs every one day/12 hours at some frequency and it pulls all the orders which have reached the terminal status from this MYSQL and puts it into Cassandra. Now how does it do that? - so you can see these two services. There's the Order Processing Service and there's a Historical Order Service. These two are again component of the whole larger Order Management System which are do certain things of their own. So Order Processing Service is the service which takes care of the whole lifecycle of the order. So once the order has been placed any changes to that order happen through this. This will also provide APId to Get the orders if somebody wants to get Order details. Again, similarly, Historical Order Service will provide all kinds of the APIs to fetch information about Historical Orders. Now Archival Service will basically query Order Processing Service to fetch all the orders who have reached terminal status, get all of those details, call Historical Order Service to insert all of those entries into its Cassandra, and once it has got a success saying I've inserted into

Cassandra, replicated in all the right places, then it will basically call Order Processing Service to delete that. Let's say there is some error, something breaks in between, it will retry the whole thing and because it is idempotent it doesn't really matter we can replay it as many number of times as we want.

Cool.

Cassandra on order archival

===========================

Now coming to the...once the user has placed the order all of the things will happen, logistics and all can happen behind the scenes. Now the user can go and see their past orders, right. That will be powered by this Orders View. So Orders View will basically...there will be a service here which kind of is the back end of

Orders View, but that'll clutter the diagram too much. Assume there's a service here which talks to the Order View, and that talks to these two services. So basically what it will do is, it will basically query Order Processing Service to fetch information about all the live orders who are in transit it. It'll call

Historical Order Service for all the orders that have been completed. It will merge the data and it will return back to display on the App or website or wherever, cool. Now why did we use a Cassandra here? - So Historical Order Service will be used in a very limited set of query patterns.

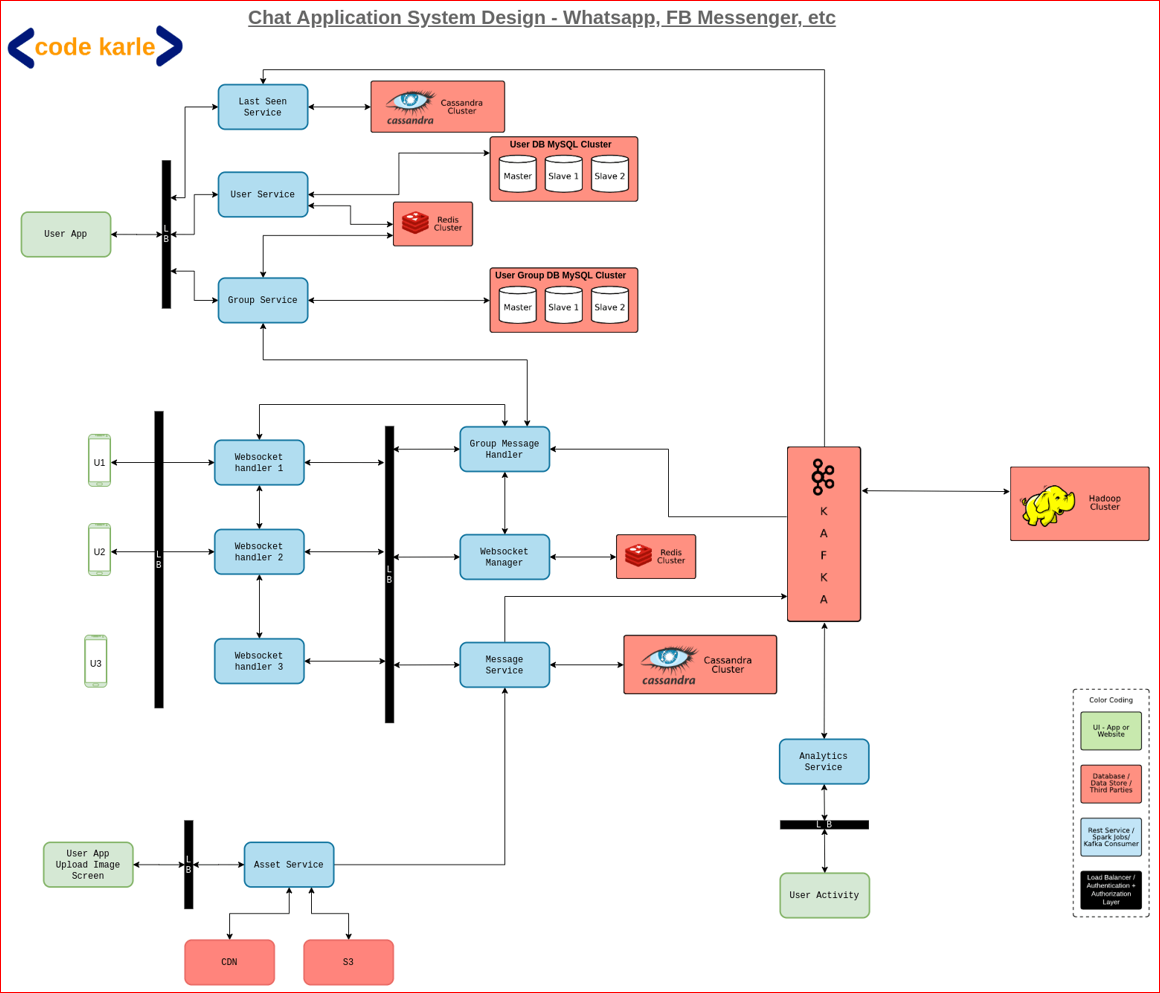
So some of the queries that will run is:

1) Get information of a particular order ID

2) Get all the orders of a particular user ID

3) possibly get all the orders of a particular seller.

6. Whatsapp System Design



WhatsApp

==============

So, in this video let's talk about how do we design a chat application, something like a WhatsApp or a Facebook Messenger or any other messenger of that sort. Now, let's look at some

Functional requirements

==========================================

We will support a One-to-One Chat wherein a person can send a message to another person or a user.

There would be something called a Group Chat where somebody could make a group of people and whenever anybody in the group sends a message to the group, that message is received by everybody in the group.

The messages could contain text, it could contain images, it could contain videos and then it could extend to other kind of documents as well.

There is something called Read Receipt, which is basically the single tick when the message is sent, double tick when the message is delivered to the

user and a blue tick when the person sees the message.

We'll also show Last Seen time of a particular user.

Non-functional requirements

===========================

This platform needs to have a very very low latency because for a chat application it needs to look very real-time so, while you are sending a message, the other person should immediately be able to see that message.

And obviously it should have a very high availability. This system should not go down, no matter what happens.

Also, there should not be any lag. In other videos that we have talked about in the past, we were able to live with a lot of lag as well. But in this video, we cannot have a lag because again, it needs to be a very real-time system wherein I am typing something to you and you can see it then and there and then respond to it.

Coming to some scale numbers, these are approximate numbers of WhatsApp. Again, I do not know if these are actually correct or not but I just read somewhere. WhatsApp has roughly 2 billion users, out of which 1.6 billion users access it on a monthly basis. These are unique users. Each day WhatsApp has somewhere close to 65 billion messages being sent across all the users. And that's a very big number.

We will try to look at how can we design a system that can deliver all of these functional requirements, with a very low latency and can live with these kind of numbers. Numbers for Facebook Messenger are approximately similar and other chat application could also be running at a very same scale. Let's look at the overall architecture of the whole system.

We'll first start with the Chatting Solution and then we'll look at other small pieces that surround it.

User flow( WebSocket Handler, WebSocket Manager, Redis ==> connection info)

============================================================================

Let's start with the User flow. All of these things that you see here are basically user devices. Could be mobile phones, could be web browsers, could be wearable devices like smart watch and all of that.

Let's say user1 wants to send a message to user2. So, the flow goes like this - user1 wants to send a message to user2 with a message id, m1. What happens is - user1 is talking to something called a WebSocket Handler 1. There will be a lot of these WebSocket Handlers. What this is - is basically, a WebSocket Handler is a server on our backend, which is keeping open connections with all the active users, right now. These are the live users, who are having an active internet connection kind of a thing. All of these users are connected to some of the machines. Then, there'll be a lot of these machines. And, all of these would be distributed across the globe. So that, people in certain geography, talk to the servers which are nearer to them, so as to have a lesser latency.

Let's just say, user1 is connected to this WebSocket Handler 1 and it sends this message to WebSocket Handler 1 saying, send this message m1 to user U2. Now, this WebSocket Handler does 2 - 3 things when this happens. It talks to something called a WebSocket Manager. WebSocket Manager is a repository of the fact that, which device is connected to what all users. Because these are probably thousands of devices, we need to keep that information at some central place, which is this WebSocket Manager Service. It sits on top of a Redis, which stores two information mainly. First of all, it stores that user1 is connected to WebSocket Handler 1

and WebSocket Handler 1 is connected to user1, user10, user20, something of that sort. So, a list of user ids that a particular machine is connected to, that information gets stored in Redis. Let's say, a connection breaks between a user and a machine. And now, the user is connected to a different machine. So, that information is updated through this WebSocket Manager, into this Redis.

Message Service

================

The second thing that WebSocket Handler 1 does is - talk to something called Message Service. Message Service is basically the repository of all the messages in the system. It will have APIs to get a message by message id, get the message by user ids and various filters. So, if you want to fetch all the unread messages by a particular user and all of that, that would be powered by this Message Service. It sits on top of a Cassandra database.

Why Cassandra? Because, again we have these billions of users, who will send many many billions of messages so, Cassandra is a fairly good data store to store that kind of a volume. Plus, the query pattern that we have fairly aligns with the query pattern that Cassandra is good at. So that's the reason we choose a Cassandra here. There are two kinds of message services that we can build. Basically there are two kind of messaging platforms that we have. One is something like a Facebook, which stores all of your messages. So, in that case everything would be stored in a Cassandra permanently. And, then there is something like a WhatsApp, which stores the messages only till the time it is not delivered to the user. Once it gets a delivery acknowledgement, then it deletes the message. So, if we want to build a WhatsApp kind of system, we could delete the message from here also. Deletes are a bit inefficient in Cassandra so, if we were to build that kind of a system, we might choose some other data store as well. But for now, Cassandra would work fairly fine.

WebSocket Handler

================

Coming back to.. we were at WebSocket Handler 1, which got this message saying U1 wants to send to U2, a message m1. Talk to both of these services and it would happen parallelly so, it can happen in any order. Message Service would save the message into Cassandra and give the message id. Let's say message id is m1. WebSocket Manager will now tell WebSocket Handler 1, saying, user U2 is connected to this WebSocket Handler 2. Now, do whatever you want to do. Now, this guy, WebSocket Handler 1 will talk to WebSocket Handler 2 saying that I have got a message for U2. I believe that guy's connected to you. Go send them that message. Now, there are 2 - 3 possibilities. WebSocket Handler 2 could actually be connected to that user still and he will send the message. Or it could not be connected. It will say that I am not connected to the user. The person has probably gone offline. So, he will not do anything and the flow ends there. Let's say, the WebSocket Handler 2 sends this message to U2. Now there are 2 - 3 things that can happen. Maybe the user is not using the app actively but the message got delivered. So, this device will send a message to WebSocket Handler 2, saying that I have received the message. Or the second option could be - the person is actually open on the chat of this user U1 and could see the message then and there. So, that device would send a message saying, I have received and seen the message m1. Now, this guy, WebSocket Handler 2 again queries WebSocket Manager, saying that this message was now either delivered or read. I need to inform user U1 so tell me which machine is talking to user U1. It will get this particular machine's IP, saying this guy is now handling user U1 and it will send this message to WebSocket Handler 1,

saying this particular message id m1, has now been read or received, whatever and go inform the user and this will be informed to the user.

How are these connections happening?

These are WebSocket connections, which is basically a bi-directional connection. So, there's no client and server here. Any party can send a message. So, that's how even server is able to send a message to the client. These are built on top of TCP. In this whole process, we queried WebSocket Manager a lot of times. User U1 calling for U2, U2 calling for U1 and all of that. Now, at the same time, if the user sends multiple messages, again, we'll try to do that. Let's try to cache this information somewhere. Each machine will cache 2 kinds of things. One it will cache is - the list of users that are connected to itself.

So, if user U2 was also connected to the same machine, it would have the data in memory. It would never have to call this manager. The other thing is - it would temporarily, for a very short duration of time, maybe 10, 20, 30 seconds, cache the information about conversations that it has had recently. So, if user U2 was connected to this handler H2, this guy (H1) will keep it in cache, saying user U2 is on this particular machine. Because if U1 sends another message to this U2, it now need not call this WebSocket Manager. But, keep in mind, this second cache needs to be stored with a very low time to live (TTL) and it should expire every 30 odd seconds. Because remember, these are global users. Let's say, if somebody is connecting from Sri Lanka, you would probably get connected to a server in India. That's a fairly large amount of distance and that communication might break in between and the person might get connected to a different machine, the next time.

So to, kind of, handle those things easily, we'll keep this cache at a very low TTL. There are another set of things. Now, the receipts that we have sent to user U1 and user U2, all those receipts.. we need to decide whether we want to store them or not. So if, let's say, we want to store the read receipts also, we can store that as a, probably, status in Cassandra, in the same table saying, this particular message has a, to start with, status as sent. Then it will move to received. Then it will move to read. We can decide.. you can decide that based on your conversation with the interviewer, whether you want to store this information or even whether you want to store the messages in the first place or not. All of those things are a conversation that you need to have with your interviewer. But, you still need to store it, for at least some time.

Let's just say that - user U1 sent a message to user U2 and then went offline. Now, the receipts wouldn't go back. So at least, for a short duration of time, those read receipts need to be stored in the database and whenever user U1 comes online, then he'll get that. Let's just say, user U1 wanted to send a message to user U3. So, a second message.. user U1 is sending a message to user U3, which is some message m2, with some text. Now, what will happen is - again, it will store in, through Message Service, it will call WebSocket Manager, to get who is the machine which is handling user U3. User U3 is offline. There is no machine that is talking to user U3. So now, WebSocket Manager will return a response saying, I don't have a machine. That flow ends there.

This WebSocket Handler 1 does not inform anybody because there is no machine talking to user U3. So, the flow breaks there. Now, when user U3 comes online, let's just say that, user U3 gets connected to this WebSocket Handler H3. What this WebSocket Handler does in the very first place is - to query Message Service and ask for - are there any messages for user U3 that are not in received or read status. This Message Service will respond back and user U3 will see all those messages. Then, when, as those messages are being received and read, those communications will flow back through the same flow like we discussed earlier. So, then, these machines will start talking to each other, for user U3 as well and those receipts, will go on to other users as well. There is a potential race condition here.

When user U1 was sending a message for user U3, assume, at the exact same time user U3 came online. We said these are parallel calls, wherein a person is figuring out the machine which is being used for user U3 and storing the message. Let's just say that, the very first thing that happened was - WebSocket Manager returned that there is no machine that is talking to user U3. The next thing that happens was - user U3 comes online and queries Message Service, saying that get me all the messages. The third thing that happens is - it stores into WebSocket Manager that this machine 3 is handling user U3. The fourth thing that happens is - this message gets stored into this Message Service. Now, WebSocket Handler 3 thinks that - it has fetched all the messages for user U3 and displayed it on the device. This WebSocket Handler 1 thinks that - user U3 is offline and I have stored the messages and done my work. But fundamentally, whatever this message m2 was, that has not been received by this guy (U3). Because, before that message was stored, this WebSocket Handler 3 retrieved all the messages.

These kind of race conditions could happen when you are doing parallel calls. There are multiple ways to handle it. You can either do it sequentially but that will increase the latencies. So, I'll not recommend that. What another solution you could possibly do is - this WebSocket Handler keeps pulling the server once in a while, saying are there any messages that I still missed. So, all of those messages could be pulled. But, that would happen at a fairly low frequency and those could be bulk calls, something like - are there any messages for user U1, U2, U3 to U100 which are in sent status, for all the active users that this machine is connected to. And, if it finds any messages, it will send all of those messages as well. There are certain things that happen on the device itself. Let's say user U1 was offline. And, you can still send message onto some user's chat, while you are offline. So, let's say, U1 sent another message to somebody while it was offline. So, what would happen is - it will try to connect. Because internet is not there, it will not be able to connect to any servers. In that scenario, what will happen.. even, in all other scenarios also.. all the messages would be stored in a local database which is persisted onto the device disk.

Let's say, it's an android application. It could use any kind of a android database like a SQLite or anything and store all those messages on the disk. Now, whenever the device gets the internet connection, it will pull those messages from a queue kind of a thing or that database and send those messages. So,

this is how it would be able to handle scenarios where the device was offline. Now, we have the next thing as group messages. How does that work? Group messages almost work in a very similar way.

Let's say, another scenario, U1 wanted to send a message to group g1, a message called m3. What would happen is - in case of groups, the behaviour is slightly different. This WebSocket Handler wouldn't take care of fanning out to all the groups. This is a very lightweight service, which just takes a message from a user and sends it to somebody or gets a message from somebody and sends it to a user. It doesn't have a lot of logic. For group messages, what it will do is - it will just send a message to Message Service, saying user U1 wants to send a message m3 to group g1, store it and do whatever you need to do. So, this message service will insert that into Cassandra saying the message is stored.

Along with that, what it will also do, for all the group messages, it will put all of those messages into a Kafka topic saying there's a group message that I've got, which is this user U1, sending to group g1, a message m3. Go send it out. From Kafka, there will be something called a Group Message Handler. This is basically a Kafka consumer, which is listening on to the same topic and it is responsible for sending out all the group messages. First of all, it does some validations that it needs to, assuming that's out of scope for us, saying whether U1 is even in the group to be able to send it to g1, but assuming that's not a part of scope. It basically queries Group Service, to get all the users that are into group g1. It gets a list of users, then it will remove the user U1 from that list.

And now it has a list of the users that are supposed to get this message. Now, it follows the exact same flow that was used for one -to- one messages but, for all of these users. So, it basically calls WebSocket Manager saying, give me all the machines that these users are connected with. It will get a response saying machine 1 is connected to user U3; U4; U5, machine 2 is connected to some other set of users and all of that. Then it will send the messages to individual machines saying, to this user send this message, to this user send this message. And then the receipts and all will follow the exact same flow that was handled earlier.

Image Upload

==============

The next thing - what happens when people are uploading assets like images or videos or anything of that sort. The very first thing that happens when you are trying to upload any content is that - there is some kind of a compression that happens. Now, in real WhatsApp, there is encryption also that happens right at device

and the encrypted messages are sent out. And when the messages are received, even the encrypted messages are received and they're decrypted but we'll keep that out of scope. We'll just think of some compression there. Let's say, U3 is sending a image to U2. So, what will happen is - U3 is sending to U2, some image. This is basically a two-step process. The first step is - upload an image to a server and get an image id and then send that image id via the regular route to user U2. And then, U2 can fetch the image. How the image upload works is - again, these are lightweight servers they wouldn't have any logic. So, very first thing is -

compression of the image, over here or video or anything. Then, it will talk to something called this Asset Service. So, this U3 is talking to Asset Service through this Load Balancer. Again, all the authentication, authorization will happen at this layer and Asset Service now gets the whole image or video or whatever. Asset Service will use Amazon S3 as a data store and it will store all the images there.

Now, based on which image is getting what kind of hits on S3, it might decide to move some of the images on CDN and it could also decide to replicate into multiple CDNs or some CDNs, from where it is getting traffic. But assuming, it has uploaded the image on S3 and it sends a response saying, let's say, it is a image i1. It will come as a regular message to U2, through this WebSocket Handler (H3).. these two WebSocket Handlers saying, send this message and the receipts will happen in the same way that we talked about earlier. There could be one optimization that can happen.

Let's say, there's a political event or a sports event and a lot of people are sharing the same image for some reason. Do we need to upload the same image multiple times, if five people are uploading the same image? Probably not!! So, before it even uploads to Asset Service, the first thing that the device will do is - take a hash of those images and send the hash content to Asset Service. Basically, it's a way of asking - do you have this image already or not? If you have it, then just send it to user U2. If you don't have it, then ask me and I'll upload it. Something of that sort.

Of course, when it's about hashing, there would be various problems like duplicate hash outputs and collisions and all of that. So, instead of taking one hash, we could probably take five different hash algorithms and use all of them to get five different hashes and if all the five hashes are matching against an image, then we'll say that, an image is already there. If it's not matching, then we'll say that, the image needs to be uploaded and it will follow the regular route that we talked about. And if it is already there, then anyway it's there, it just sends a message.

User Service.

============

Now, let's look at remaining few components and how they help in the whole system. So, this is the user app, which talks to all of these components. It could be a browser as well. The very first thing is - something called User Service. This is basically a service, which owns the user profile information. It could be - their name, their user id, their profile picture in case of WhatsApp and some user preferences. In case of Facebook, it will be a larger set of data. But all of that, whatever that data is, it would be residing in a clustered MySQL, which runs the user database. All of those information would also be cached in Redis cluster,

very similar to the way we have implemented in other similar solutions. Against a key, we'll be having the user profile information and a lot of other information as well.

Group Service

============

There's something called Group Service, which basically maintains all the group information. It basically has information of - which group has what all users. Think of it as a many-to-many table, with a user id and a group id and a time when user was added into the group and maybe a status saying, whether the user is admin or something of that sort. Again, this information would be stored in the MySQL cluster, which would be geographically distributed and it will have multiple slaves through which the read queries will be run. It will also have a Redis sitting on top of it. So, all the APIs like, let's say, get users in a group

and something of that sort, would be powered by this Redis cluster and those APIs would be sitting within the Group Service. Each time, either User Service or Group Service, queries Redis and it does not find the data, is when they actually query one of the salves of these databases, fetch the information, put it into the Redis and respond back to whoever is calling them. And then, group service was used in the group messaging workflow, that we looked at earlier. Each action that a user is taking on their app, gives us a lot of information.

Analytics

==========

For example, if a person is visiting some user's profile a lot of times, that tells us that the person is interested in this user. It could be used for some kind of Analytics. Let's say, if a particular person is talking a lot about sports, that does tell us that the person is a sports enthusiast. How could we do that? All events that are happening, either they are events within the app or they are messages or anything of that sort, would be sent either to this Analytics Service and then to Kafka, or directly to Kafka, in case of messages. There could be a Spark Streaming Consumer running on top of this, which could do a lot of inferences. First of all, it could do some message classification into some tags, like a message is a sports kind of a message or a general message or a politics kind of a message and then try to classify users basis that. So, that is one kind of analytics.

Another thing could be, just simply, some information about what are people talking about, in general. This is not possible if the messages are end to end encrypted. But if that's not the case, then we should be easily able to infer all of that.

There are two parts. One is - doing some logic, based on some streaming content. That could be done by a simple Spark Streaming Consumer. And, that consumer would also put that data into Hadoop. And then, you can run a lot of analytics queries, based on Hadoop. Something like, which are the people who talk about sports a lot or which are the people who just talk about a particular kind of political party or anything of that sort. Those kind of analytics could be run out of that Hadoop cluster. These events are also having one more usage.

Last Seen Service

==================

Each time a person does an activity on the app, those events are listened by something called Last Seen Service. This Last Seen Service is basically, the service used to store the last seen time of a user. This service.. I made it to sit on top of a Cassandra database, which has a column called user\_id and a column called last\_seen\_time. This could also alternatively sit on top of a Redis, if you have finite amount of data. But along with last seen time, if you want to store certain other information, then Redis might have memory issues. So, in that scenario, you can use a Cassandra. You might have a doubt that - when we use MySQL for user information, why don't we use a MySQL here? The reason is - let's say, a user is on the app. Every five seconds, the app will send a notification that the user is live.

So, every five seconds, for each user, this will have a insert into the database. What that means is - this is having an extremely high throughput and MySQL might not be able to support that because these are all the writes, which will go into the master node of MySQL. But, if it's a Cassandra, it could be spread out easily across a lot of users. Redis might also have those update problem because for all the users, you are doing a lot of inserts. So, Redis might not be able to scale. But, if it's a Cassandra, it should easily be able to scale to any amount of traffic, any amount of users and those things coming at any frequency.

One more thing - There are two kinds of events that are spawned. One are the events that are spawned by the app and one are the events that are spawned by the users. Example of events spawn by app could be - as soon as you turn the internet on, the app would make a WebSocket connection

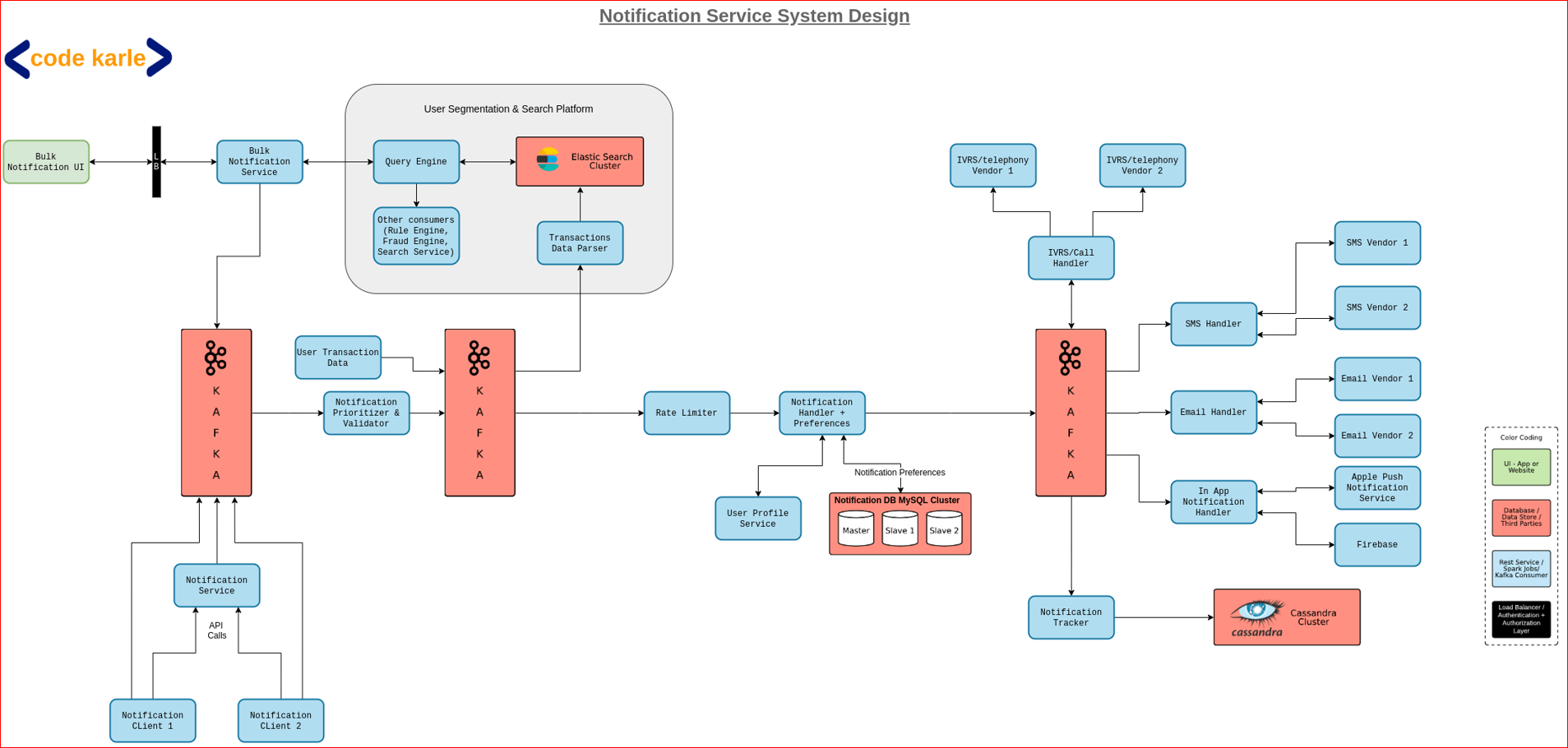
and pull in some requests. That is not a user activity. So, that would not be captured as a, you know, activity of a user in the Last Seen Service. But, as soon as the person opens the app or closes the app or does anything within the context of an app, those are actually the events that Last Seen Service would be looking at. In general, all the services that we looked at, we are looking at here, in the other previous section, all of those are, as usual, horizontally scalable. So, when your number of users or anything increases, you just add more number of nodes and that should be able to handle itself.

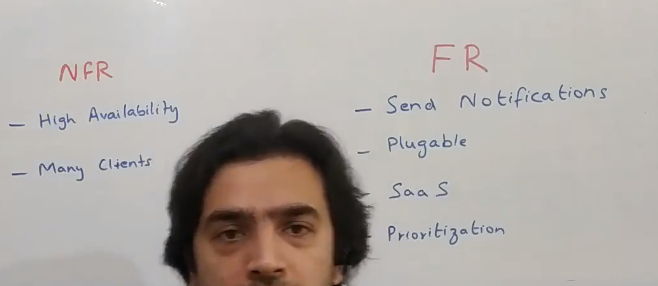
Other than that, you will obviously need to do a lot of monitoring, alerting - basically to ensure that you are able to get to your NFRs of no lag, low latency and high availability.

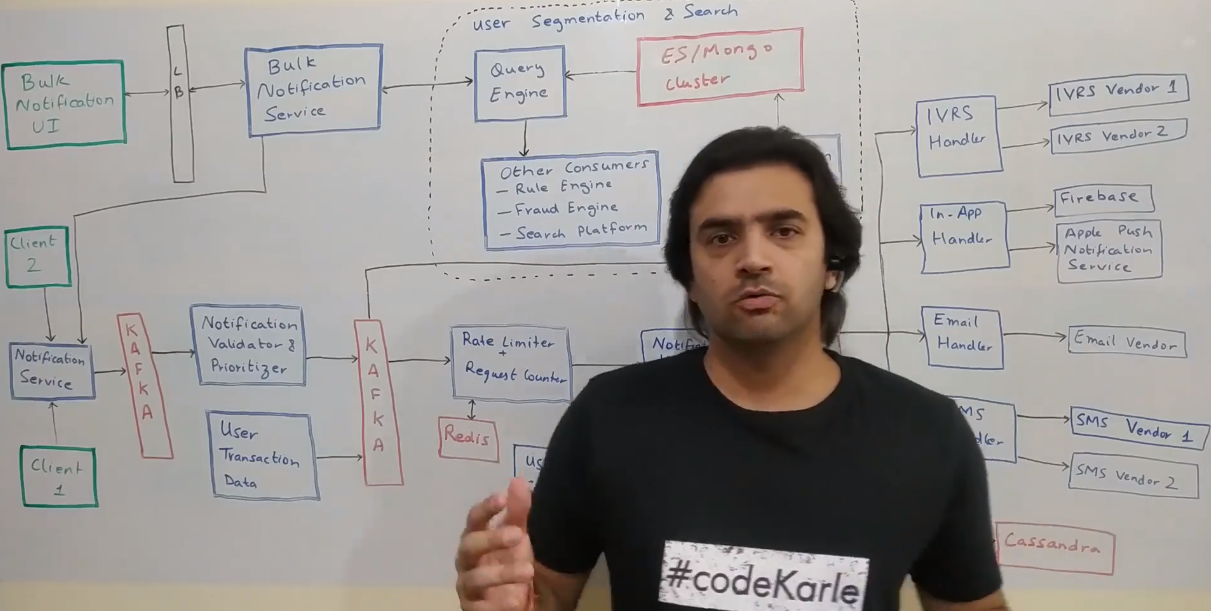
So, some of the important things that you want to, kind of, keep a track of from a monitoring perspective is - the CPU memory utilization of your services, your disk utilization in the databases, your throughputs and latency onto your web services and the disk utilization in your Kafka and more importantly the lag in Kafka. Let's say, if your lag increases, you need to add more number of consumers. So, all of those metrics you can capture, store it into a Grafana kind of a store and plot some graphs out of it. And also, you can build some alerting out of it. In case your numbers go beyond a certain threshold, you can send out alerts. Those alerts could be used to make some decisions. For example, if your throughput is increasing significantly,

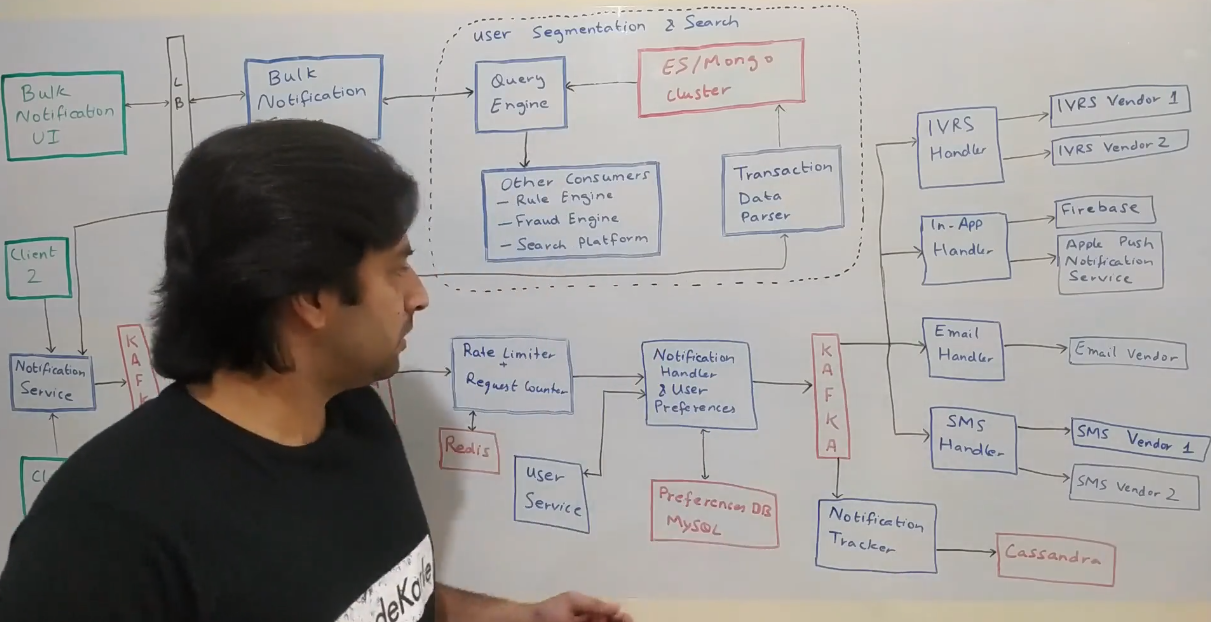
you could add more machines into a particular service and in fact those things could be automated as well. So, if you're using AWS kind of a thing, you might write a script which looks at your throughput and it adds more nodes or removes certain nodes. All of those things are very important when it comes to a chat application because one of your very critical NFR is that it needs to be actually real time. If it's not real time, then it will give a very bad experience and then people will kind of drop out of your platform. So yeah!! That should be it about a chat application.

7. Notification System at scale









Notification Service

=====================

But there will be a small number of type of queries that will run on this Cassandra and Cassandra very good when you have that finite number of queries(by type) but a large set of data. that's the reason we have used Cassandra. Now coming to whenever an order is placed we want to notify the customer that your order has been successfully placed. It will be delivered to in 3 days or something of that sort. You'll have to possibly notify a seller, you might have to notify somebody else

or when something happens. Let's say an order is cancelled by the seller you want to notify the customers. So all of those notifications would be taken care by this Notification Service. Think of it as an abstraction which can abstract out all various kinds of notification like SMSs, Emails and all of that. I made

another video(on Notification Service Design) which talks about in detail of how do you implement Notification Service. So you can go have a look at that as well. While ... a user is placing all the orders, all of those things are happening, all the events are going into this Kafka on which we are running a Spark Streaming

Consumer. It does a lot of things. One of the... very first things that it does is it can publish a report saying in the last one hour what are our items which have been ordered the most. Or which category has generated the most amount of revenue like electronics or food or something of that sort, right. So

it can publish out some of the reporting metrics that we want to quickly look at. Other than that, it will also put the whole of the data into Hadoop cluster. Now we have real order information of a user which is a very good thing for recommendations So on top of it, will have some spark jobs which will run some very standard ALS kind of algorithms, which would be predicting that this particular user, given that he's ordered certain things in the past, what are the next set

of things that this person might order. Or because this customer looks like that customer and that customer ordered so-and-so product, so this customer might also order so-and-so product. So there could be, a lot of kind of models that you can run to predict what are the right set of recommendations for

this user, which would be then send to this Recommendation Service which we looked at in the last section, it will store it into its Cassandra and the next time when the user comes in, we'll show them those kind of recommendations. It would not just have like recommendations based on your orders but also similarity, so again

taking the same example if you have ordered a white board you will definitely want to order marker. So all of those things would also be taken care by this.

----------------------------------------------------------------

4 - Notification Service System Design

==================================

Hello, everyone! In this video, we'll look at how can we design a notification service that is scalable enough. This is never a stand alone system but this is always embedded in some other system design. Even I have used it in a lot of other system designs videos that I have made. Let's say if you are building a e-commerce application or a booking system or anything of that sort. You'll always have a notification service which will be used to notify your consumers.

Functional (FRs)

===============

Let's look at how can we build a notification service. Now let's look at some Functional (FRs) and Non Functional Requirements (NFRs)

that this platform should support.

The very obvious first thing is that it should be able to send notifications.

The next thing is that it should be pluggable. Now what does pluggable mean? So let's say we want to support SMS and email as one form of notification. Now, tomorrow somebody might want to say that I want to support in-app notifications as well. So it should be easy enough to add that. It can be further extended to a lot of kinds of notifications. So, for example, you could have something like a WhatsApp notification or anything of that sort. So, extendability is something that should be taken care of.

The next thing is that it should be built as a SaaS product. Why SaaS? because the main idea is you should know who's sending what number of notifications.

And it should be possible for you to rate limit. There are two use cases where you'll need that. So rate limiting as a offering you'll need, if you are giving out to other companies as a product to use. But, even internally, you would need to do some kind of rate limiting. Let's just say its being used by a company like Amazon. Now there are multiple business verticals. If all of them send you notifications, then you'll end up getting hundreds of notifications in a day. So that's a very bad user experience. So notification system should be able to overall put a rate limit across all the users across all the platforms saying a particular user should not get more than five notifications in a day. There could be certain amount of classifications done. So there are two kinds of notifications. One is a transactional notification saying you have made an order, this much amount of money has been deducted from your account, something of that sort. So transactional notifications are ok to get. But promotional notifications should always have a rate limit.

So if you're building for within your company, then you'll build a user level rate limiting. But if you're giving it out externally to other companies as a product, then you might want to put a rate limiting saying how many requests can you make for this server. Or maybe make billing tiers and price them accordingly. So basically this is something that you need to capture at least and then act upon.

The next thing is prioritization. Again... we'll support multiple priorities of messages. So the idea of priorities is that some messages are low priority and certain messages are high priority. Let's say, if you are sending a one time password (OTP), then that's a very high priority message. Why? Because if the user wouldn't get that SMS or email or anything, then they would not log into your platform and they'll not do the transaction that they wanted to do. But if it's a promotional message, it is okay if it's low priority and if it gets delayed, if it goes to the user let's say half an hour late also, it doesn't really matter.

So we'll have this prioritization, basis which you'll always process the high priority messages first and then the low priority messages.

non functional requirements

===========================

Coming to the non functional requirements (NFRs). This platform should always be available. Why? Because, if you are planning to build it as a SaaS product which can be used by other companies, then down time would really cost us a lot. Plus, it should be built in a way that it's easy enough to add more clients.

And whenever we want, we should be able to attribute saying how many clients have made us how many number of requests. Now, let's look at the overall architecture of the whole system. Just a disclaimer to start with, if you are building it for a small enough use case like let's just say if you want to send out some email notifications to some customers, given some criteria, you can build all of this as one deployable service and put all the logic in one place. But if you truly want to build it as a SaaS product wherein enormous number of clients would be using it for a lot of notifications, then this kind of a system would probably be worth it.

Let's just look at the starting point. The starting point of the whole system is then a couple of clients i.e. Client 1, Client 2, could be any number of clients, who want to send out a notification. Now, there are 2-3 kinds of requests that they can send you. But all of those requests would come into something called as a Notification service, which is an interface for us to talk to the other teams in the company, other companies or anything that we want. There are two kinds of request mainly. One is where they tell you that I want to send this particular content to this particular user, let's say a email id kind of a thing, as an e-mail.

This is one of the requests. Other request could be saying, I have this user id and send them this notification. And you decide how do you want to send, whether as an email or as a SMS or whatever? Normally the first kind of a model would be used by other companies where they want to decide that they need to send as SMS or email or whatever. The second model would generally be used when you are building this to be consumed within your own company. But normally as a SaaS product, its good to have both the interfaces. The idea of notification service is, for most kind of requests, it will just take the request, put it into Kafka and respond back to the client saying, I have taken the request and I will send the notification in a couple of seconds at max.

You could make this transaction as a synchronous flow but that will take a bit more time and that will keep the client blocked. So, it's probably best to take the request and dump it into a Kafka and move on. But let's just say if for a very critical scenario, you need it to be a synchronous flow. You can basically make the whole process synchronous through API calls. But assuming its a Kafka for now, let's look at what happens next. This Notification service would just do a very basic set of validations saying that email id should not be null or user id should not be null, content should not be null, something of that sort. There might be a lot of other validations, given certain kinds of events that you want to do. All of those validations would happen in this component which is called Notification Validater and Prioritizer. It does a couple of things. Validations is one part of it. The main thing that it does, it decides the priority for a message.

So based on some attribute within the message, let's say, we''ll keep a message type identifier kind of a thing in a request and based on that, it will decide the priority of a message. Normally, OTP is something which will be high priority because if the user can't login into the system, they can't place an order and things like that. The second most high priority thing would be a transactional message saying your order has been placed, it will be delivered by so and so time. The third, least priority one would would a promotional message wherein you are sending out offers and coupons and stuff like that. So, it decides the priority and puts the event into a Kafka topic, specific for each priority. So there'll be a topic for high priority messages, there'll be a different topic for medium priority, there'll be a different topic for low priority. And while consuming, the consumers will first consume the high priority messages and then the medium and low priority. The idea is - we don't want any lag in the high priority messages, but we should be at times okay if there's a spike on low priority message and if it takes time, that's okay.

Rate Limiter

============

Now comes something called as a Rate Limiter. These two components can be interchanged depending upon the conversation with your interviewer. But I'm keeping Rate Limiter before it because the next component is a slightly heavier component.

Rate Limiter does two kinds of rate limiting.

1) It checks that the client who's calling me, is that client allowed to call me these many number of times. That is the first thing.

2) The user whom I'm sending this notification, am I supposed to send this notification to this user these many number of times.

Let's go into detail. So there could be a subscription that we have with a certain client that they can call us maybe 10 times in a second. All of those kind of rate limiting, would be one kind of rate limiting. The other thing is there could be a configuration saying I cannot send more than three promotional messages to a particular user in a day. That would be another kind of rate limiting. Both these rate limiting would be implemented in a very similar way. Saying there would be a key. It could be a client id, user id, anything. And whenever you get a request, you basically increment that key into this Redis for a certain timestamp.

Or maybe the timestamp need not be a exact timestamp, it could be day, it could be second, it could be at a minute level or any level. The moment you exceed that threshold is when you drop that request. There's one more thing that it does, which is called request counting. Let's just say with some of the clients you have a pay for use kind of a subscription model, saying they can call you any number of times, there is no rate limiting as such but as the number of calls increases, the amount of money we charge them also increases. So they pay at a request basis. For those kind of clients, we'll implement it in a very similar way. Just that we will not restrict the request. We'll keep incrementing the counter and then there can be reporting built on top of it.

Notification Handler and User Preferences.

=========================================

The next component is something called as a Notification Handler and User Preferences. First let's look at what are user preferences. User might give us a preference saying don't send me SMS, send me emails instead. Or a user could have said that I want to unsubscribe from all your promotional messages. So all of those would be handled by this User Preference Service. It has two components that it talks to. One is a Preferences DB, which is a single source of truth for all the user preferences that we have in our system. This would be mainly used when we are building it for our own use. Let's say Amazon is building this notification system for their own use. This will normally not be used when you're integrating with third parties. Third parties would handle this piece on their end. The second thing it will talk to is a User service which is basically - let's say you got a request saying to this user id 123, send this particular text. What is that user 123? It's something that you can search from user service to get the email id or phone number or anything of that sort, given a user id. This basically could handle one more kind of rate limiting. A user might say that don't send me more than three promotional messages in a week or one promotional message in a week. So, if that kind of a rate limiting also exists and want to support that, then this whole Rate Limiting module would come after this Notification Handler. But, because this is a very light weight thing and this calls a DB, calls an external service, that's the reason I have kept it after the Rate Limiter. But based on your conversation with your

interviewer, you could swap either of these.

Notification Handler.

====================

Once we have all of this finalized, we basically have a full fledged system that we can actually use to notify.

We have a phone number, we know that we'll send it as an SMS and we have the content or we have the email id, we know we'll send it as an e-mail and we have the content. All of that is put into another Kafka for basically sending it out. Why do we need so many Kafkas? Because let's just say we had a lot of SMSes and this SMS service is not able to handle it. One way is to increase the number of consumers. But if there are certain spikes in certain clients, it's probably not worth to add a hardware for all the time for that. So we might as well put into Kafka and this SMS Handler could handle things at its own pace. There are multiple handlers that sit on top of this Kafka. You could decide to build all of these as one deployable unit, but again, with the SaaS (Software as a Service) thing in mind,

I'll rather keep that as a separate service in each of them. So there are multiple handlers. Let's take a couple of examples. So all the SMS requests that will come into this Kafka would be handled by this SMS Handler. Plus, there could be different topics for each of these. So email could have one topic, SMS could have one topic and stuff like that. Now SMS Handler might integrate with multiple vendors for sending SMSes. Let's say you are a global company. You have one vendor that deals with SMSes in Asia, another vendor that deals with SMSes in USA , another vendor for Europe, stuff like that. So you could have multiple vendors that you are integrated with and all the SMS vendor integration could be done through this SMS Handler. Now, let's say if you have... if you are in India and you have 5 vendors that work in India. So what request should go to which vendor? What is the priority? How do we distribute request amongst them? That piece could also be handled within this own handler. Same thing, you have for emails. You have a Email Handler. And by the way, this communication with the handler and vendor could be a sync call.

Moving back to Email Handler. Email Handler takes all the e-mail requests and calls the email vendor which will send out the real emails. This email vendor could be a simple SMTP server that you have at your end. Then you will have In-App Handler, which basically handles all the in-app notifications that you want to

send out or a push notification, kind of a thing. You can use a Firebase for Android and probably use a Apple push notification service for sending out such notifications on iOS platform. You could also have a IVRS Handler. A classic example would be, I don't know, either Amazon or Flipkart, one of them. When you place a very high value cash on delivery (CoD) order, they actually send you an IVRS call, saying that we have got this order from you, do you, are you really sure you have kept this as a cash on delivery order and you can press 1 or 2 to kind of confirm or reject. So all of those things could be handled by the IVRS Handler.

This is a very low throughput scenario and it would happen once in a while when compared to SMS or emails or anything of that sort. You can again scale these independently based on the kind of traffic you have.

pluggability

==============

Now coming to pluggability. Let's say somebody comes and says that you need to add WhatsApp aslo as a notification mechanism. So basically you just need to add a WhatsApp Handler and basically introduce a type called WhatsApp Messaging at this layer and all of that can flow through all the way. This Kafka... this basically Notification Handler will have a new topic for WhatsApp messages and a WhatsApp Handler will basically take care of sending those notifications out. So that's how you can make it fairly pluggable.

Notification Tracker

====================

Now coming to what this Notification Tracker is? You always need to keep a track of what all notifications you have sent. In case somebody sues you or there is some

auditing required, you need to know what all communication you have sent out. So for all of that, you have a Notification Tracker which puts all the notifications that you have sent out on its own Cassandra. This is normally a write only thing, which would be read once in a while when there's an audit or something of that sort.

Depending upon your traffic and the throughput, you could decide to club some of these components. So for example, you can decide to club all of these components as a single deployable service. You could also club both these components into a single deployable service and you can move the Prioritizer right there and make lesser number of components and try to club all of this as one service as well if it's a very low throughput thing. You can decide it based on your conversation with your interviewer on how do you want to do that? Now, let's do something different.

bulk notifications

==================

What if you want to send bulk notifications? What if you want to do something like - for all the users who have ordered a TV in last 24 hours, send them a notification like installation service or anything of that sort. Or for all the users who had ordered milk 3 days back, send them a notification that whether do you want to order the same thing again or not? So all of those things come under something called as a bulk notification. Now, how do you want to do that?

So the very first thing is there would be a UI kind of a thing, which is represented as Bulk Notification UI, which will talk to something called as a Bulk Notification service, which takes a filter criteria and a notification and sends it out.

Filter criteria would be something like find all the users who have placed a milk order, anything in last three days and send them a notification with certain text.

How does that work? So there's something called as this User Transaction Data. This basically abstracts out a lot of services which are outside the scope of this notification service but real business functional services. For example, again, taking the example of Amazon, let's say, there are a couple of services that handle

their e-commerce orders. There are a couple of services that handle their pantry orders. All of those services would be putting their transactional information - that your order has been placed, it has been delivered, it has been canceled - all of those kind of attributes into various Kafka topics. What we could do is basically build a search functionality on top of those topics. Normally, in most companies, you do have those kind of products already built. But assuming you don't have that, let's try to build that. And if we have that, we can just leverage it. What will happen is - this will put information into a lot of Kafkas, lot of topics. I am abstracting it out as this one Kafka. All of those information would be listened by something called as a Transaction Data Parser. Now, this Transaction Data Parser basically... first of all, parses all of that information. Now, because you have multiple clients and multiple services who are sending data, they could be putting in multiple formats of json or XML. So, for example, pantry (Amazon) could have a separate signature and regular e-commerce could have a separate signature. This data parser basically takes all of those messages, puts it in a format or converts it in a format that it understands and then it puts it into its own data store.

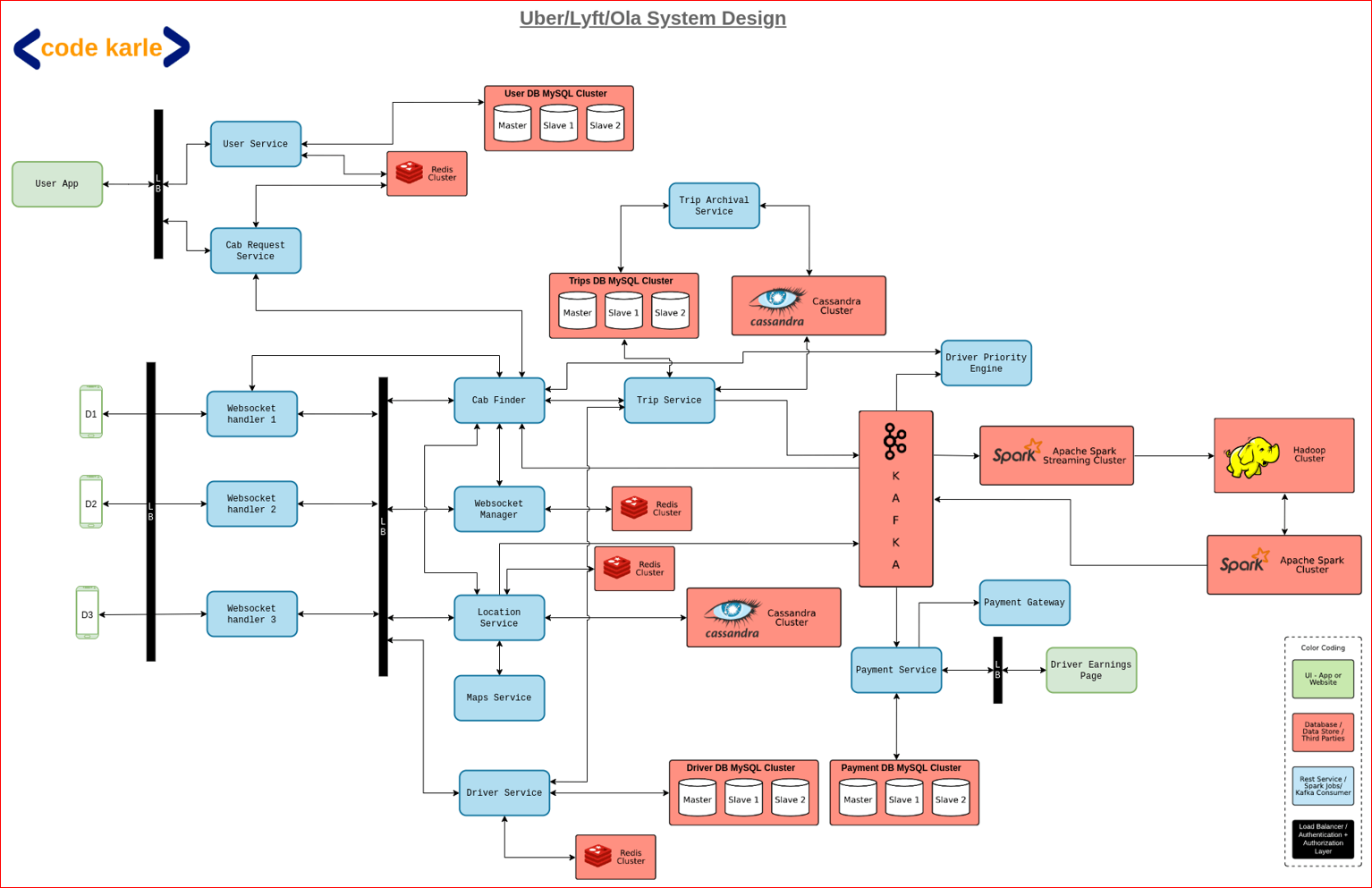
It could be Elasticsearch or a MongoDB kind of a structure where you could do complicated aggregations and nested queries. On top of this data store, there's something called as a Query Engine. That Query Engine basically takes a query, which could be like an aggregation plus a filter kind of a thing, saying, find out all the users who are in Bangalore let's say, who have ordered some food item in the last few days, or find all the people who are having some items in their cart but have not placed the order or any kind of a filter criteria that you want to build that would be powered by this Query Engine. It would have its own DSL (Domain Specific Language), which would be a format - how do you kind of structure your query and that would be a signature that it exposes. Now, this Query Engine would query that thing on this data store. So it also understands the schema of this data store and it basically would return us a list of users that match that criteria.

Now this Bulk Notification service has a list of users that it has got from Query Engine, it has a message that it got from this UI, it basically call this Notification service saying send this notification to all of these users and these users, it would get from this Query Engine. This Query Engine is not something which would just be used by this Notification service. It could be used by a lot of services. You could have a very simple Rule Engine which possibly says that if a user has cancelled 10 consecutive orders in last 3 days, go deactivate their account. They are probably, you know, just doing fraud. Or, there could be a simple Fraud Engine running which has some criteria and takes some action basis that. All these condition and action kind of a thing, wherever you have a condition running

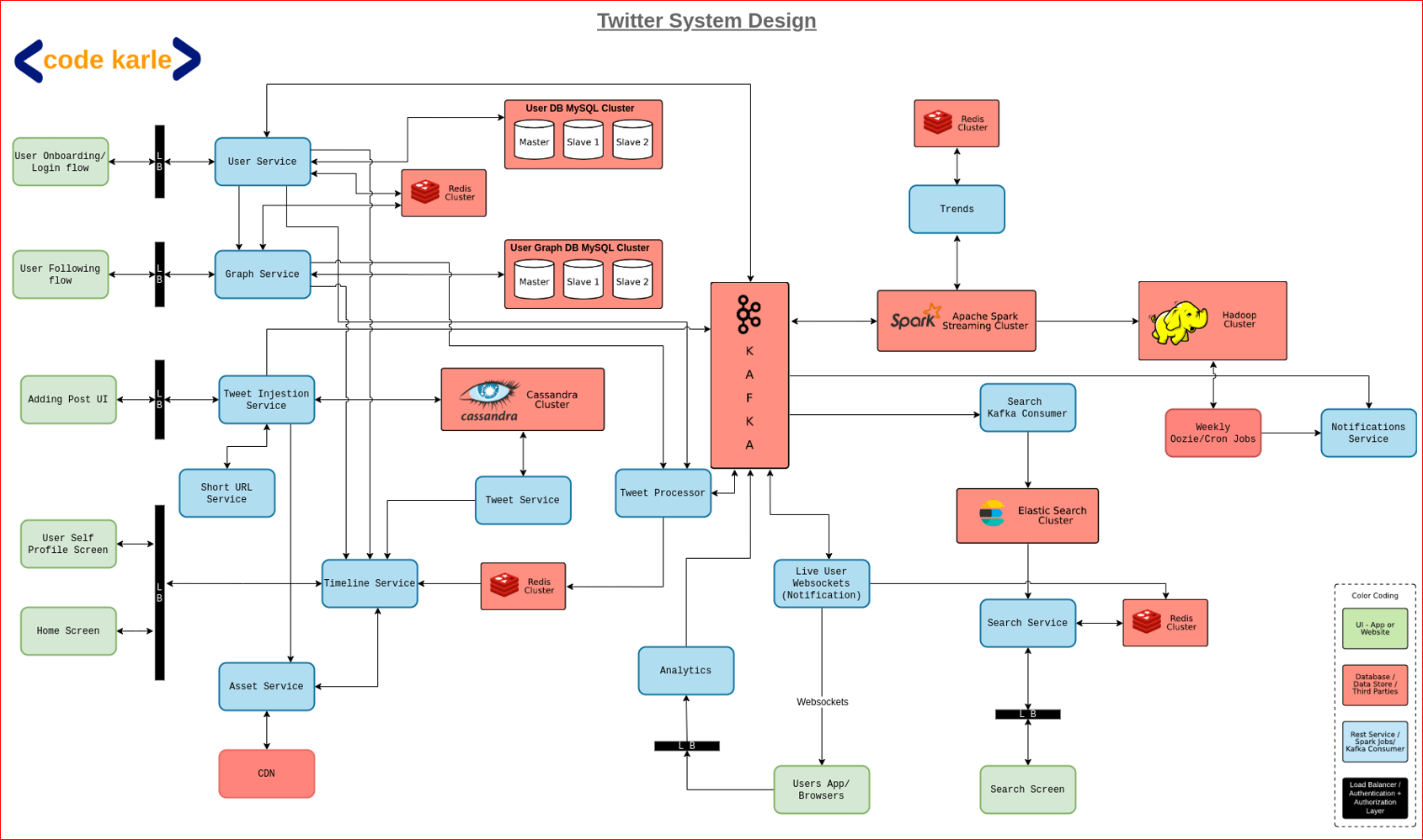
and action happening, all of those could be, you know, powered by this kind of a thing. In fact, your Notification service itself is that kind of a thing. So you have a filter criteria as your condition and action is basically the notification sending process. So this could also be a part of your Rule Engine. Something like 5 days after an order has been delivered, send a mail to consumers for giving a product review. So something of that sort. And this could then also be used to build a Search platform altogether. But overall, this is something that and if this is something that is already built in the company, you could just leverage that and

have this Bulk Notification service talk to the querying platform of that whole platform.

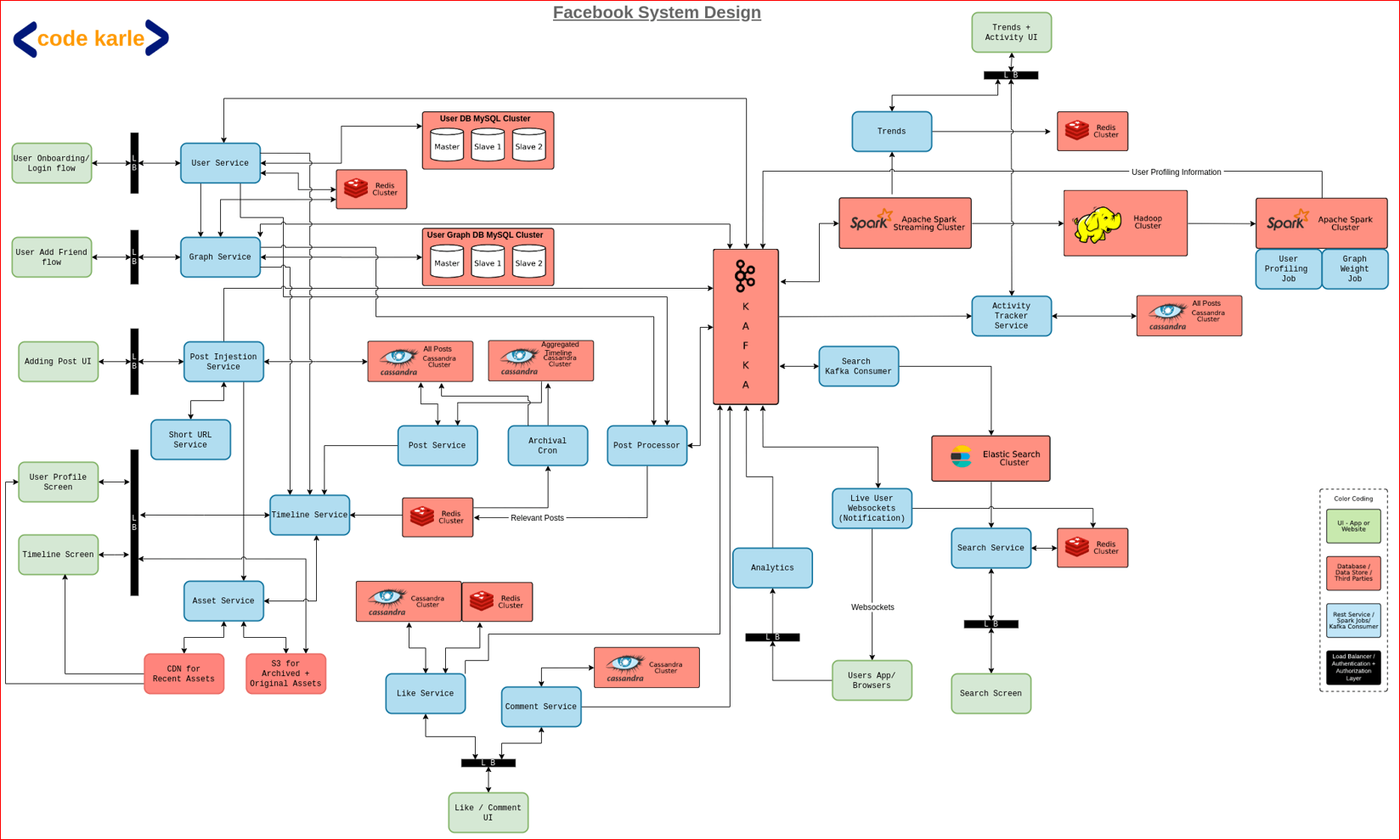
8. Uber System Design



9. Twitter System Design



10. Facebook / Instagram System Design



11. YouTube / Netflix System Design

